TION PAGE

Form Approved
OPM No. 0704-0188

Public reporti needed, and Headquarter Management

AD-A242 525

ponse, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data timate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington avis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENC TE

1991

3. REPORT TYPE AND DATES COVERED
FINAL

4. TITLE AND SUBTITLE
Ada QUALITY AND STYLE: GUIDELINES FOR PROFESSIONAL PROGRAMMERS

5. FUNDING NUMBERS

6. AUTHOR(S)

DTIC
ELECTE
NOV1 1991
S C D

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SOFTWARE PRODUCTIVITY CONSORTIUM, INC.
SI) BUILDING
2214 ROCK HILL RD
HERNDON, VA 22070-4005

8. PERFORMING ORGANIZATION REPORT NUMBER
SPC-91061-N VERSION 02.00.02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada JOINT PROGRAM OFFICE
THE PENTAGON, RM. 3E118
WASHINGTON, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
UNCLASSIFIED - UNLIMITED PUBLIC DISTRIBUTION

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

THE SECOND EDITION HAS BEEN PRODUCED TO CORRECT, CLARIFY, AND ENHANCE VARIOUS TOPICS IN THE ORIGINAL VERSION. INSTANTIATIONS HAVE BEEN INCLUDED WITHIN RELEVANT SECTIONS. WHILE THERE HAVE BEEN CHANGES, ADDITIONS AND DELETIONS TO THE GUIDELINES, THE MAJOR CONCEPTUAL CHANGES HAVE BEEN IN THE FOLLOWING AREAS: USE OF OTHERS CLAUSE IN CASE STATENEBTS, USE OF WHILE LOOPS AND BLOCKS; EXCEPTION HANDLING, ANONYMOUS TASK TYPES, AND CONDITIONAL AND TIMED ENTRY CALLS. THE SECTIONS ON COMMENTS AND THE CHAPTER ON REUSE HAVE BEEN EXPANDED. THIS IS THE SUGGESTED Ada STYLE GUIDE FOR USE IN DoD PROGRAMS BY THE Ada JOINT PROGRAM OFFICE.

14. SUBJECT TERMS
Ada STYLE GUIDE

15. NUMBER OF PAGES
180

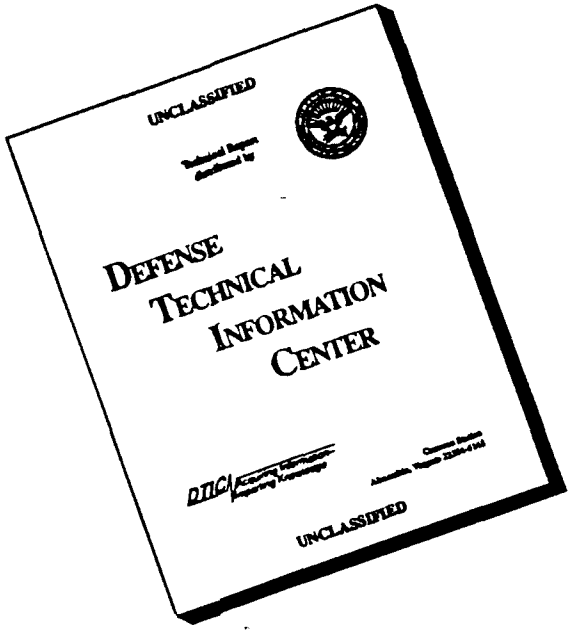
16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCL		

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Ada Quality and Style:
Guidelines for Professional Programmers

SPC-91061-N

VERSION 02.00.02

1991

Software Productivity Consortium
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070-4005

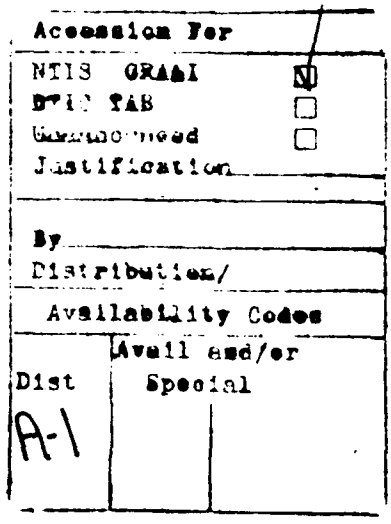
Unlimited Distribution

Ada Quality and Style:
Guidelines for Professional Programmers

SPC-91061-N

VERSION 02.00.02

1991



PREFACE

The second edition has been produced to correct, clarify, update and enhance various topics in the original version. The book has undergone minor restructuring. While the chapters remain the same, some of the sections have been rearranged for clarity. Also, instantiations have been included within relevant sections. In some sections, there is an additional subsection on automation. While there have been changes, additions and deletions to the guidelines, the major conceptual changes have been in the following areas: use of the others clause in case statements, use of while loops and blocks, exception handling, anonymous task types, and conditional and timed entry calls. The sections on comments and the chapter on reuse have been expanded. Additionally, some examples have been enhanced.

We invite comments on this guidebook to continue enhancing its quality and usefulness. We will consider suggestions for current guidelines as well as areas for future expansion. Examples that highlight particular points are most helpful.

Please direct comments to:

Technology Transfer Division - AQS
Software Productivity Consortium
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070-4005
(703) 742-7211

Please fill out and mail the registration form at the back of this document to receive updates and other information.

AUTHORS AND ACKNOWLEDGEMENTS

The authors for the second edition are Kent Johnson, Elisa Simmons, and Fred Stluka. Contributors are Alex Blakemore and Robert Hofkin. Reviewers include Alex Blakemore, Rick Conn, Tim Harrison, Dave Nettles, and Doug Smith. Additional support has been provided by Vicki Clatterbuck, Leslie Hubbard, and Debra Morgan.

The following people contributed to an instantiation of the first edition's guidelines: Rich Bechtold, Pete Bloodgood, Shawna Gregory, Tim Powell, Dave Nettles, Kevin Schaan, Doug Smith and Perry Tsacoumis.

Special thanks are extended to Loral for providing feedback in the form of their Software Productivity Laboratory Ada Standards.

The Consortium would also like to acknowledge those involved in the first edition. The authors were Richard Drake, Samuel Gregory, Margaret Skalko, and Lyn Uzzle. Managing the project was Paul Cohen. The contributors and reviewers were Mark Dowson, John Knight, Henry Ledgard, and Robert Mathis. Word processing was performed by Debra Morgan.

Additional supporters included Bruce Barnes, Alex Blakemore, Terry Bollinger, Charles Brown, Neil Burkhard, William Carlson, Susan Carroll, John Chludzinski, Vicki Clatterbuck, Robert Cohen, Elizabeth Comer, Daniel Cooper, Jorge Diaz-Herrera, Tim Harrison, Robert Hofkin, Allan Jaworski, Edward Jones, John A.N. Lee, Eric Marshall, Charles Mooney, John Moore, Karl Nyberg, Arthur Pyster, Samuel Redwine, Jr., William Riddle, Lisa Smith, Fred Stluka, Kathy Velick, David Weiss, and Howard Yudkin.

CONTENTS

CHAPTER 1
Introduction

This book is intended to help the computer professional produce better Ada programs. It presents a set of specific guidelines for using the powerful features of Ada in a disciplined manner. Each guideline consists of a concise statement of the principles that should be followed, and a rationale explaining why following the guideline is important. In most cases, an example of the use of the guideline is provided, and in some cases a further example is included showing the consequences of violating the guideline. Possible exceptions to the application of the guideline are explicitly noted, and further explanatory notes, including notes on how the guideline could be automated by a tool, are provided where appropriate. Many of the guidelines are specific enough to be adopted as corporate or project programming standards. Others require a managerial decision on a particular instantiation before they can be used as standards. In such cases, a sample instantiation is presented and used throughout the examples. Such instantiations should be recognized as weaker recommendations than the guidelines themselves. These issues are discussed in Section 1.4 of this introduction. Other sections of the introduction discuss how this book should be used by various categories of software development personnel.

Ada was designed to support the development of high-quality, reliable, reusable, portable software. For a number of reasons, no programming language can ensure the achievement of these desirable objectives on its own. For example, programming must be embedded in a disciplined development process that addresses requirements analysis, design, implementation, verification, validation and maintenance in an organized way. The use of the language must conform to good programming practices based on well established software engineering principles. This book is intended to help bridge the gap between these principles and the actual practice of programming in Ada.

Clear, readable, understandable source text eases program evolution, adaptation, and maintenance. First, such source text is more likely to be correct and reliable. Second, effective code adaptation is a prerequisite to code reuse, a technique that has the potential for drastic reductions in system development cost. Easy adaptation requires a thorough understanding of the software, this is facilitated considerably by clarity. Finally, since maintenance (really evolution) is a costly process that continues throughout the life of a system, clarity plays a major role in keeping maintenance costs down. Over the entire life cycle, code has to be read and understood far more often than it is written; the investment of writing readable, understandable code is thus well worthwhile. Many of the guidelines in this book are designed to promote clarity of the source text.

There are two main aspects of code clarity. Careful and consistent layout of the source text on the page or the screen can enhance readability dramatically. Similarly, careful attention to the structure of code can make it easier to understand. This is true both on the small scale (e.g., by careful choice of identifier names or by disciplined use of loops), and on the large scale (e.g., by proper use of packages). Both layout and structure are treated by these guidelines.

Comments in source text is a controversial issue. There are arguments both for and against the view that comments enhance readability. The biggest problem with comments in practice is that people often fail to update them when the associated source text is changed, thereby making the commentary misleading. Commentary should be minimized and largely reserved for highlighting cases where there are overriding reasons to violate one of the guidelines. If possible, source text should use self-explanatory names for objects and program units, and use simple, understandable program structures so that little additional commentary

is needed. The extra effort in selecting (and entering) appropriate names, and the extra thought needed to design clean and understandable program structures are fully justified.

Programming texts often fail to discuss overall program structure; Chapter 4 addresses this. The majority of the guidelines in that chapter are concerned with the application of sound software engineering principles such as information hiding and separation of concerns. The chapter is neither a textbook on nor an introduction to these principles; rather it indicates how they can be realized using the features of Ada.

A number of other guidelines are particularly concerned with reliability and portability issues. They counsel avoidance of language features and programming practices that either depend on properties not defined in Ada or on properties that may vary from implementation to implementation. Some of these guidelines, such as the one forbidding dependence on expression evaluation order, should never be violated. Others may have to be violated in special situations such as interfacing to other systems. This should only be done after careful deliberation, and such violations should be prominently indicated. Performance constraints are often offered as an excuse for unsafe programming practices; this is rarely a sufficient justification.

Software tools could be used to enforce, encourage, or check conformance to many of the guidelines. At present, such tools for Ada primarily consist of code formatters or syntax directed editors. Existing code formatters are often parameterizable and can be instantiated to lay out code in a way consistent with many of the guidelines in this book.

This book is intended for those involved in the actual development of software systems written in Ada. Below, discusses how to make the most effective use of the material presented. Readers with different levels of Ada experience and different roles in a software project will need to use the book in different ways. Specific comments to three broad categories of software development personnel are addressed: inexperienced Ada programmers, experienced Ada programmers, and software development managers.

1.1 HOW TO USE THIS BOOK

There are a number of ways in which this book can be used: as a reference on good Ada style; as a comprehensive list of guidelines which will contribute to better Ada programs; or as a reference work to consult about using specific features of the language. The book contains many guidelines, some of which are quite complex. Learning them all at the same time should not be necessary; it is unlikely that you will be using all the features of the language at once. However, it is recommended that all programmers (and, where possible, other Ada project staff) make an effort to read and understand Chapters 2, 3, and 4 and Chapter 5 up to Section 5.7. Some of the material is quite difficult (for example, Section 4.2 which discusses visibility) but it covers issues which are fundamental to the effective use of Ada, and is important for any software professional involved in building Ada systems.

The remainder of the book covers relatively specific issues. Exceptions and erroneous execution is covered at the end of Chapter 5, and tasking, portability and reuse is covered in Chapters 6, 7, and 8 respectively. You should be aware of the content of this part of the book. You may be required to follow the guidelines presented in it, but you could defer more detailed study until needed. Meanwhile, it can serve as useful reference material about specific Ada features; for example, the discussion of floating point numbers in the chapter on portability.

Chapter 9 is directed at software project managers. It repeats those guidelines that need to be instantiated to be used as standards, and indicates the instantiation that has been adopted in the guidelines' examples. Chapter 10 consists of an extended example of an Ada program that conforms to the guidelines presented.

This book is not intended as an introductory text on Ada or as a complete manual of the Ada language. It is assumed that you already know the syntax of Ada, and have a rudimentary understanding of the semantics. With such a background, you should find the guidelines useful, informative, and often enlightening.

If you are learning Ada you should equip yourself with a comprehensive introduction to the language such as (Barnes 1989) or (Cohen 1986). The Ada Language Reference Manual (Department of Defense 1983) should be regarded as a crucial companion to this book. The majority of guidelines reference the sections of the Ada Language Reference Manual that define the language features being discussed. Appendix A cross references sections of the Ada Language Reference Manual to the guidelines.

Throughout the book, references are given to other sources of information about Ada style and other Ada issues. The references are listed at the end of the book, followed by a bibliography which includes them and other relevant sources consulted during the book's preparation.

1.2 TO THE NEW Ada PROGRAMMER

At first sight, Ada offers a bewildering variety of features. It is a powerful tool intended to solve difficult problems and almost every feature has a legitimate application in some context. This makes it especially important to use Ada's features in a disciplined and organized way. The guidelines in this book forbid the use of few Ada features. Rather, they show how the features can be systematically deployed to write clear, high-quality programs. Following the guidelines will make learning Ada easier and help you to master its apparent complexity. From the beginning, you can write programs that exploit the best features of the language in the way that the designers intended.

Programmers experienced in using another programming language are often tempted to use Ada as if it were their familiar language, but with irritating syntactic differences. This pitfall should be avoided at all costs, it can lead to convoluted code that subverts exactly those aspects of Ada that make it so suitable for building high-quality systems. You must learn to "think Ada"; following the guidelines in this book and reading the examples of their use will help you to do this as quickly and painlessly as possible.

To some degree, novice programmers learning Ada have an advantage. Following the guidelines from the beginning helps in developing a clear programming style that effectively exploits the language. If you are in this category, it is recommended that you adopt the guidelines for those exercises you perform as part of learning Ada. Initially, developing sound programming habits by concentrating on the guidelines themselves, and their supporting examples, is more important than understanding the rationale for each guideline. Note that each chapter ends with a summary of the guidelines it contains.

1.3 TO THE EXPERIENCED Ada PROGRAMMER

As an experienced programmer you are already writing code that conforms to many of the guidelines in this book. In some areas, however, you may have adopted a personal programming style that differs from that presented here, and you might be reluctant to change. Carefully review those guidelines that are inconsistent with your current style, make sure that you understand their rationale, and consider adopting them. The overall set of guidelines in this book embodies a consistent approach to producing high-quality programs which would be weakened by too many exceptions.

Another important reason for general adoption of common guidelines is consistency. If all the staff of a project write source text in the same style, many critical project activities are easier. Consistent code simplifies formal and informal code reviews, system integration, within-project code reuse and the provision and application of supporting tools. In practice, corporate or project standards may require deviations from the guidelines to be explicitly commented, so adopting a nonstandard approach may require extra work.

1.4 TO THE SOFTWARE PROJECT MANAGER

Technical management plays a key role in ensuring that the software produced in the course of a project is correct, reliable, maintainable, and portable. Management must create a project-wide commitment to the production of high-quality code; define project-specific coding standards and guidelines; foster an understanding of why uniform adherence to the chosen coding standards is critical to product quality; and establish policies and procedures to check and enforce that adherence. The guidelines contained in this book can aid such an effort.

An important activity for managers is the definition of coding standards for a project or organization. These guidelines do not, in themselves, constitute a complete set of standards, but can serve as a basis for them. A number of guidelines indicate a range of decisions, but do not prescribe a particular decision. For example, the second guideline in the book (Guideline 2.1.2) advocates using a consistent number of spaces for indentation, and indicates in the rationale that 2 to 4 spaces would be reasonable. With your senior technical staff, you should review each such guideline and arrive at a decision about its instantiation that will constitute your project or organizational standard. To support this process, Chapter 9 of the book lists all guidelines that need instantiation to be used as standards. It also gives a possible instantiation for each guideline that corresponds to the decision adopted by this book, and used in the extended example of Chapter 10.

Two other areas require managerial decisions about standardization. Guideline 3.1.4 advises you to avoid arbitrary abbreviations in object or unit names. You should prepare a glossary of acceptable abbreviations for a project that allows the use of shorter versions of application-specific terms (e.g., FFT for Fast Fourier Transform or SPN for Stochastic Petri Net). You should keep this glossary short and restrict it to terms which

need to be used frequently as part of names. Having to refer continually to an extensive glossary to understand source text makes it hard to read.

The portability guidelines given in Chapter 7 need careful attention. Adherence to them is important even if the need to port the resulting software is not currently foreseen. Following the guidelines improve the potential reusability of the resulting code in projects that use different Ada implementations. You should insist that when particular project needs force the relaxation of some of the portability guidelines, nonportable features of the source text are prominently indicated. Observing the Chapter 7 guidelines requires definition and standardization of project- or organization-specific numeric types to use in place of the (potentially nonportable) predefined numeric types.

Your decisions on standardization issues should be incorporated in a project or organization coding standards document.

With coding standards in place, you need to ensure adherence to them. Probably the most important aspect of this is gaining the wholehearted commitment of your programming staff to use them. Given this commitment, and the example of high-quality Ada being produced by your programmers, it will be far easier to conduct effective formal code reviews that check compliance to project standards.

Consistent coding standards work well with automatic tool support. If you have a tools group in your project or organization, they can be tasked to acquire or develop tools to support your standards. It is very cost effective to use tools to enforce standards. Where tools cannot be used to automatically modify code to conform to standards, they can often be used to at least check conformance. See the automation notes sections associated with many of the guidelines.

Some general issues concerning the management of Ada projects are discussed by (Foreman and Goodenough 1987).

CHAPTER 2
Source Code Presentation

The physical layout of source text on the page or screen has a strong effect on its readability. This chapter contains source code presentation guidelines intended to make the code more readable.

In addition to the general purpose guidelines, specific recommendations are made in the "instantiation" sections. If you disagree with the specific recommendations, you may want to adopt your own set of conventions which still follow the general purpose guidelines. Above all, be consistent across your entire project.

An entirely consistent layout is hard to achieve or check manually. Therefore you may prefer to automate layout with a tool for parameterized code formatting, or incorporate the guidelines into an automatic coding template. Beware, however, that such tools are limited. Some of the guidelines and specific recommendations presented in this section cannot be enforced by a formatting tool because they are based on the semantics, not the syntax, of the Ada code. More details are given in the "automation notes" sections below.

2.1 CODE FORMATTING

The "code formatting" of Ada source code affects how the code looks, not what the code does. Topics included here are horizontal spacing, indentation, alignment, pagination, and line length. The most important guideline is to be consistent throughout the compilation unit as well as the project.

2.1.1 Horizontal Spacing

guideline

- Use consistent spacing around delimiters.
- Use the same spacing as you would in regular prose.

instantiation

Specifically, leave at least one blank space in the following places, as shown in the examples throughout this book. More spaces may be required for the vertical alignment recommended in subsequent guidelines.

- Before and after the following delimiters and binary operators:

```
+      -      *      /      &
<      =      >      /=     <=     >=
:=     =>     |      ..
:
<>
```

- Outside of the quotes for string (") and character (') literals, except where prohibited below.
- Outside, but not inside, of parentheses.
- After commas (,) and semicolons (;).

Do not leave any blank spaces in the following places, even if this conflicts with the above recommendation.

- After the plus (+) and minus (-) signs when used as unary operators.
- Inside of label delimiters (<< >>).
- Before and after the following:
  . **
- Between multiple consecutive opening or closing parentheses.
- Before commas (,) and semicolons (;).

example

```
REGISTER (PC) := REGISTER (A);

OPERATOR_PRECEDENCE_MNEMONICS : STRING := "My Dog Ain't Smart,"
                                        & " but he obeys"
                                        & " My Dear Aunt Sallie.";

ARRAY_NAME (INDEX) := MEMORY (BASE_ADDRESS + (INDEX * ELEMENT_LENGTH));

GET_NEXT_VALUE (SENSOR);

type SIGNED_WHOLE_16 is range -(2**15) .. (2**15) - 1;
```

rationale

It is a good idea to use whitespace around delimiters and operators because they are typically short (one or two character) sequences which can easily get lost among the longer keywords and identifiers. Putting whitespace around them makes them stand out. Consistency in spacing also helps by making the source code easier to scan visually.

However, many of the delimiters (commas, semicolons, parentheses, etc.) are familiar as normal punctuation marks. It is distracting to see them spaced differently in a computer program than in normal text. Therefore, they should be spaced the same (no spaces before commas and semicolons, no spaces inside of parentheses, etc.).

exception

The one notable exception to this is the colon (:). In Ada, it is useful to use the colon as a tabulator, or a column separator (see Guideline 2.1.4). In this context, it makes sense to put spaces before and after the colon, rather than only after as in normal text.

automation notes

The guidelines in this section are easily enforced with an automatic code formatter.

2.1.2 Indentation

guideline

- Indent and align nested control structures, continuation lines, and embedded units consistently.
- Distinguish between indentation for nested control structures and for continuation lines.
- Use spaces for indentation, not the tab character (Nissen and Wallis 1984, §2.2).

instantiation

Specifically, the following indentation conventions are recommended, as shown in the examples throughout this book. Note that the minimum indentation is described. More spaces may be required for the vertical alignment recommended in subsequent guidelines.

- Use the recommended paragraphing shown in the Ada Language Reference Manual (Department of Defense 1983).
- Use three spaces as the basic unit of indentation for nesting.
- Use two spaces as the basic unit of indentation for continuation lines.

A label is outdented three spaces. A continuation line is indented two spaces:

```
<<label>>                          | <long statement with line break>
   <statement>                     |   <trailing part of same statement>
```

The if statement and the plain loop:

```
if <condition> then                | <name>:
   <statements>                    |    loop
elsif <condition> then             |       <statements>
   <statements>                    |       exit when <condition>;
else                               |       <statements>
   <statements>                    |    end loop;
end if;                            |
```

Loops with the for and while iteration schemes:

```
<name>:                            | <name>:
   for <scheme> loop               |    while <condition> loop
      <statements>                 |       <statements>
   end loop;                       |    end loop;
```

The block and the case statement as recommended in the Ada Language Reference Manual (Department of Defense 1983):

```
<name>:                            | case <expression> is
   declare                         |    when <choice> =>
      <declarations>               |       <statements>
   begin                           |    when <choice> =>
      <statements>                 |       <statements>
   exception                       |    when others =>
      when <choice> =>             |       <statements>
         <statements>              | end case;
      when others =>               |
         <statements>              |
   end <name>;                     |
```

These case statements save space over the the Ada Language Reference Manual (Department of Defense 1983) recommendation and depend on very short statement lists, respectively. Whichever you choose, be consistent.

```
case <expression> is               | case <expression> is
when <choice> =>                   |    when <choice> => <statements>
     <statements>                  |                     <statements>
when <choice> =>                   |    when <choice> => <statements>
     <statements>                  |    when others   => <statements>
when others =>                     | end case;
     <statements>                  |
end case;                          |
```

The various forms of selective wait and the timed and conditional entry calls:

```
select                             | select
   when <guard> =>                 |    <entry call>;
      <accept statement>           |    <statements>
      <statements>                 | or
or                                 |    delay <interval>;
      <accept statement>           |    <statements>
      <statements>                 | end select;
or                                 |
   when <guard> =>                 |
      delay <interval>;            |
      <statements>                 |
or                                 | select
   when <guard> =>                 |    <enter call>;
      terminate;                   |    <statements>
else                               | else
   <statements>                    |    <statements>
end select;                        | end select;
```

The accept statement and a subunit:

```
accept <specification> do          |    separate (<parent unit>)
   <statements>                    | <proper body>
end <name>;                        |
```

Body stubs of the program units:

```
procedure <specification> is
   separate;

function  <specification>
  return <type> is
   separate;
```

```
package body <name> is
   separate;

task body <name> is
   separate;
```

Proper bodies of program units:

```
procedure <specification> is
   <declarations>
begin
   <statements>
exception
   when <choice> =>
      <statements>
end <name>;

function  <specification>
  return  <type name> is
   <declarations>
begin
   <statements>
exception
   when <choice> =>
      <statements>
end <name>;
```

```
package body <name> is
   <declarations>
begin
   <statements>
exception
   when <choice> =>
      <statements>
end <name>;

task body <name> is
   <declarations>
begin
   <statements>
exception
   when <choice> =>
      <statements>
end <name>;
```

Context clauses on compilation units are arranged as a table and are indented so as not to obscure the introductory line of the unit itself. Generic formal parameters do not obscure the unit itself. Function, package, and task specifications use standard indent:

```
   with <name>,
        <name>,
        <name>;
   use  <name>,
        <name>,
        <name>;
<compilation unit>


generic -- <kind of unit> <name>
   <formal parameters>
<compilation unit>
```

```
function  <specification>
  return <type>;

package <name> is
   <declarations>
private
   <declarations>
end <name>;

task type <name> is
   entry <declaration>
end <name>;
```

Instantiations of generic units, and indentation of a record:

```
procedure <name> is
   new <generic name> <actual>

function <name> is
   new <generic name> <actuals>

package <name> is
   new <generic name> <actuals>
```

```
type ... is
   record
      <component list>
      case <discriminant name> is
         when <choice> =>
            <component list>
         when <choice> =>
            <component list>
      end case;
   end record;
```

Indentation for record alignment:

```
for <name> use
   record <alignment clause>
      <component clause>
   end record;
```

example

```
loop

   if INPUT_FOUND then
      COUNT_CHARACTERS;
   else
      RESET_STATE;
      DEFAULT_STRING := "This is the long string returned by"
                      & " default.  It is broken into multiple"
                      & " Ada source lines for convenience."
      CHARACTER_TOTAL := (FIRST_PART_TOTAL  * FIRST_PART_SCALE_FACTOR )
                       + (SECOND_PART_TOTAL * SECOND_PART_SCALE_FACTOR)
                       + DEFAULT_STRING'length
                       + DELIMITER_SIZE;
   end if;

end loop;
```

rationale

Indentation improves the readability of the code because it gives the reader a visual indicator of the structure of the program. The levels of nesting are clearly identified by indentation and the first and last keywords in a construct can be matched visually.

While there is much discussion on the number of spaces to indent, the reason for indentation is code clarity. The fact that the code is indented consistently is more important than the number of spaces used for indentation.

Additionally, the Ada Language Reference Manual (Department of Defense 1983, §1.5) says that the layout shown in the examples and syntax rules in the LRM is the recommended code layout to be used for Ada programs. "The syntax rules describing structured constructs are presented in a form that corresponds to the recommended paragraphing. ... Different lines are used for parts of a syntax rule if the corresponding parts of the construct described by the rule are intended to be on different lines. ... It is recommended that all indentation be by multiples of a basic step of indentation (the number of spaces for the basic step is not defined)."

It is important to indent continuation lines differently from nested control structures to make them visually distinct. This prevents them from obscuring the structure of the code as the user scans it.

Indenting with spaces is more portable than indenting with tabs because tab characters are displayed differently by different terminals and printers.

exception

According to the Ada Language Reference Manual (Department of Defense 1983, §1.5), "... On the other hand, if a complete construct can fit on one line, this is allowed in the recommended paragraphing."

automation notes

The guidelines in this section are easily enforced with an automatic code formatter.

2.1.3 Alignment of Operators

guideline

- Align operators vertically to emphasize local program structure and semantics.

example

```
if SLOT_A >= SLOT_B then
   TEMPORARY := SLOT_A;
   SLOT_A    := SLOT_B;
   SLOT_B    := TEMPORARY;
end if;

NUMERATOR   := (B**2) - (4 * A * C);
DENOMINATOR := 2 * A;
SOLUTION_1  := -B + SQUARE_ROOT (NUMERATOR / DENOMINATOR);
SOLUTION_2  :=  B + SQUARE_ROOT (NUMERATOR / DENOMINATOR);
```

```
X :=   A * B
     + C * D
     + E * F;

Y :=   (A * B) + C    -- basic equation
     + (2 * D) - E    -- account for ...
     - 3.5;           -- error factor
```

rationale

Alignment makes it easier to see the position of the operators and therefore, puts visual emphasis on what the code is doing.

The use of lines and spacing on long expressions can emphasize terms, precedence of operators, and other semantics. It can also leave room for highlighting comments within an expression.

exceptions

If vertical alignment of operators forces a statement to be broken over two lines, and especially if the break is at an inappropriate spot, it may be preferable to relax the alignment guideline.

automation notes

The last example above shows a type of "semantic alignment" which is not typically enforced or even preserved by automatic code formatters. If you break expressions into semantic parts and put each on a separate line, beware of using a code formatter later. It is likely to move the entire expression to a single line and accumulate all the comments at the end. However, there are some formatters which are intelligent enough to leave a line break intact when the line contains a comment. With such a formatter, the layout in the last example above could not be generated automatically, but could be preserved while other formatting was done.

2.1.4 Alignment of Declarations

guideline

- Use vertical alignment to enhance the readability of declarations.
- Provide at most one declaration per line.

example

Variable and constant declarations can be laid out in a table with columns separated by the symbols :, :=, and --

```
PROMPT_COLUMN : constant         := 40;
QUESTION_MARK : constant STRING := " ? ";   -- prompt on error input
PROMPT_STRING : constant STRING := " ==> ";
```

If this results in lines which are too long, they can be laid out with each part on a separate line with its unique indentation level.

```
INPUT_LINE_BUFFER
   : USER_RESPONSE_TEXT_FRAME
      := (others => ' ');
         -- If the declaration needed a comment, it would fit here.
```

Declarations of enumeration literals can be listed in one or more columns as:

```
type OP_CODES is
  (PUSH,
   POP,
   ADD,
   SUBTRACT,
   MULTIPLY,
   DIVIDE,
   SUBROUTINE_CALL,
   SUBROUTINE_RETURN,
   BRANCH,
   BRANCH_ON_ZERO,
   BRANCH_ON_NEGATIVE);
```

or, to save space:

```
type OP_CODES is
  (PUSH,              POP,                  ADD,
   SUBTRACT,          MULTIPLY,             DIVIDE,
   SUBROUTINE_CALL,   SUBROUTINE_RETURN,    BRANCH,
   BRANCH_ON_ZERO,    BRANCH_ON_NEGATIVE);
```

or, to emphasize related groups of values:

```
type OP_CODES is
  (PUSH,             POP,
   ADD,              SUBTRACT,           MULTIPLY,           DIVIDE,
   SUBROUTINE_CALL,  SUBROUTINE_RETURN,
   BRANCH,           BRANCH_ON_ZERO,     BRANCH_ON_NEGATIVE);
```

rationale

Many programming standards documents require tabular repetition of names, types, initial values, and meaning in unit header comments. These comments are redundant and can become inconsistent with the code. Aligning the declarations themselves in tabular fashion (see the examples above) provides identical information to both compiler and reader, enforces at most one declaration per line, and eases maintenance by providing space for initializations and necessary comments. A tabular layout enhances readability, thus preventing names from "hiding" in a mass of declarations. This applies to type declarations as well as object declarations.

automation notes

Most of the guidelines in this section are easily enforced with an automatic code formatter. The one exception is the last enumerated type example, which is laid out in rows based on the semantics of the enumeration literals. An automatic code formatter will not be able to do this, and will likely move the enumeration literals to different lines.

2.1.5 More on Alignment

guideline

- Align parameter modes and parentheses vertically.

instantiation

Specifically it is recommended that you:

- Place one formal parameter specification per line.
- Vertically align parameter names, colons, the reserved word `in`, the reserved word `out`, and parameter types.
- Place the first parameter specification on the same line as the subprogram or entry name. If any of the parameter types are forced beyond the line length limit, place the first parameter specification on a new line indented as for continuation lines.

example

```
procedure DISPLAY_MENU (TITLE   : in     STRING;
                        OPTIONS : in     MENUS;
                        CHOICE  :    out ALPHA_NUMERICS);
```

or

```
procedure DISPLAY_MENU_ON_PRIMARY_WINDOW
  (TITLE   : in     STRING;
   OPTIONS : in     MENUS;
   CHOICE  :    out ALPHA_NUMERICS),
```

or

```
procedure DISPLAY_MENU (
  TITLE   : in     STRING;
  OPTIONS : in     MENUS;
  CHOICE  :    out ALPHA_NUMERICS
  );
```

Aligning parentheses makes complicated relational expressions more clear:

```
if (FIRST_CHARACTER not in ALPHA_NUMERICS) or else
   (not VALID_OPTION (FIRST_CHARACTER)    )          then
```

rationale

This facilitates readability and understandability. Aligning parameter modes provides the effect of a table with columns for parameter name, mode, type, and if necessary, parameter-specific comments. Vertical alignment of parameters across subprograms within a compilation unit increases the readability even more.

note

Various options are available for subprogram layout. The second example above aligns all of the subprogram names and parameter names in a program.This has the disadvantage of occupying an unnecessary line where subprogram names are short, and looking awkward if there is only one parameter.

The third example above is a format commonly used to reduce the amount of editing required when parameter lines are added, deleted, or reordered. The parentheses don't have to be moved from line to line. However, the last parameter line is the only one without a semicolon.

The last example above shows the alignment of a multiple condition if statement. The alignment emphasizes the variables that are tested and their relationships. The `or else` is by itself so the major connective operator is not lost in the expression. This helps the reader to parse it.

automation notes

Most of the guidelines in this section are easily enforced with an automatic code formatter. The one exception is the last example above, which shows vertical alignment of parentheses to emphasize terms of an expression. This is difficult to achieve with an automatic code formatter unless the relevant terms of the expression can be determined strictly through operator precedence.

2.1.6 Blank Lines

guideline

- Use blank lines to group logically related lines of text (NASA 1987).

example

```
if ... then

   for ... loop
      ...
   end loop;

end if;
```

This example separates different kinds of declarations with blank lines:

```
type EMPLOYEE_RECORD is
   record
      NAME          : NAME_STRING;
      DATE_OF_BIRTH : DATE;
      DATE_OF_HIRE  : DATE;
      SALARY        : MONEY;
   end record;

type DAY is
  (MONDAY,       TUESDAY,
   WEDNESDAY,    THURSDAY,
   FRIDAY,       SATURDAY,
   SUNDAY);

subtype WEEKDAY is DAY range MONDAY    .. FRIDAY;
subtype WEEKEND is DAY range SATURDAY  .. SUNDAY;
```

rationale

When blank lines are used in a thoughtful and consistent manner, sections of related code are more visible to readers.

automation notes

Automatic formatters do not enforce this guideline well because the decision on where to insert blank lines is a semantic one. However, many formatters have the ability to leave existing blank lines intact. Thus, you can manually insert the lines and not lose the effect when you run such a formatter.

2.1.7 Pagination

guideline

- Highlight the top of each package or task specification, the top of each program unit body, and the begin and end statements of each program unit.

instantiation

Specifically, it is recommended that you:

- Use a line of dashes, beginning at the same column as the current indentation.
- Use the shorter of the two dashed lines if they are adjacent.
- Omit the dashed line above the `begin`.
- When putting a dashed line at the top of a compilation unit, put it before, not after, the context clauses.

example

```
------------------------------------------------------------------------------
with BASIC_TYPES;
package body SPC_NUMERIC_TYPES is
   ---------------------------------------------------------------------------
   function MAX (LEFT  : in TINY_INTEGER;
                 RIGHT : in TINY_INTEGER)
     return TINY_INTEGER is
   begin
      if (LEFT > RIGHT) then
         return LEFT;
      else
         return RIGHT;
      end if;
   end MAX;
   ---------------------------------------------------------------------------
   function MIN (LEFT  : in TINY_INTEGER;
                 RIGHT : in TINY_INTEGER)
     return TINY_INTEGER is
   begin
      if (LEFT < RIGHT) then
         return LEFT;
      else
         return RIGHT;
      end if;
   end MIN;
   ---------------------------------------------------------------------------
   ...
   ---------------------------------------------------------------------------
begin
   MAX_TINY_INTEGER := MIN (SYSTEM_MAX, LOCAL_MAX);
   MIN_TINY_INTEGER := MAX (SYSTEM_MIN, LOCAL_MIN);
   ...
end SPC_NUMERIC_TYPES;
------------------------------------------------------------------------------
```

rationale

It is easy to overlook parts of program units that are not visible on the current page or screen. The page lengths of presentation hardware and software vary widely. By clearly marking the program's logical page boundaries (e.g., with a dashed line), you enable a reader to check quickly whether all of a program unit is visible. Such pagination also makes it easier to scan a large file quickly, looking for a particular program unit.

note

This guideline does not address code layout on the physical "page" because the dimensions of such pages vary widely and no single guideline is appropriate.

exception

If a unit contains very few declarations, then the visual distance between the top of the unit and its "begin" is very small. In such a case, the dashed line above the "begin" is unnecessary.

automation notes

The guidelines in this section are easily enforced with an automatic code formatter.

2.1.8 Number of Statements Per Line

guideline

- Start each statement on a new line.
- Write no more than one simple statement per line.
- Break compound statements over multiple lines.

example

```
-- Use
if END_OF_FILE then
   CLOSE_FILE;
else
   GET_NEXT_RECORD;
end if;

-- rather than
if END_OF_FILE then CLOSE_FILE; else GET_NEXT_RECORD; end if;

-- exceptional case
PUT ("A="); PUT (A); NEWLINE;
PUT ("B="); PUT (B); NEWLINE;
PUT ("C="); PUT (C); NEWLINE;
```

rationale

A single statement on each line enhances the reader's ability to find statements and helps prevent statements being missed. Similarly, the structure of a compound statement is clearer when its parts are on separate lines.

note

A source statement is any Ada language statement that is terminated with a semicolon. If the statement is longer than the remaining space on the line, continue it on the next line. This guideline includes declarations, context clauses, and subprogram parameters.

According to the Ada Language Reference Manual (Department of Defense 1983, §1.5), "The preferred places for other line breaks are after semicolons".

exceptions

The example of PUT and NEWLINE statements shows a legitimate exception. This grouping of closely related statements on the same line makes the structural relationship between the groups clear.

automation notes

The guidelines in this section are easily enforced with an automatic code formatter, with the single exception of the last example above which shows a semantic grouping of multiple statements onto a single line.

2.1.9 Source Code Line Length

guideline

- Adhere to a maximum line length limit for source code (Nissen and Wallis 1984, §2.3).

instantiation

Specifically, it is recommended that you:

- Limit source code line lengths to a maximum of 78 characters.

rationale

When Ada code is ported from one system to another, there may be restrictions on the record size of source line statements, possibly for one of the following reasons: some operating systems may not support variable length records for tape I/O; some printers and terminals support an 80-character line width with no line-wrap. We recommend 78 because even 80 character terminals often have trouble with lines which are exactly 80 characters long. Leaving a two character margin avoids boundary problems which can cause spurious line wrap, etc.

Source code must sometimes be published for various reasons, and letter-size paper is not as forgiving as a computer listing in terms of the number of usable columns.

In addition, there are human limitations in the width of the field of view for understanding at the level required for reading source code. These limitations correspond roughly to the 70 to 80 column range.

automation notes

The guidelines in this section are easily enforced with an automatic code formatter.

2.2 SUMMARY

- Use consistent spacing around delimiters.
- Use the same spacing as you would in regular prose.
- Indent and align nested control structures, continuation lines, and embedded units consistently.
- Distinguish between indentation for nested control structures and for continuation lines.
- Use spaces for indentation, not the tab character.
- Align operators vertically to emphasize local program structure and semantics.
- Use vertical alignment to enhance the readability of declarations.
- Provide at most one declaration per line.
- Align parameter modes and parentheses vertically.
- Use blank lines to group logically related lines of text.
- Highlight the top of each package or task specification, the top of each program unit body, and the begin and end statements of each program unit.
- Start each statement on a new line.
- Write no more than one simple statement per line.
- Break compound statements over multiple lines.
- Adhere to a maximum line length limit for source code.

CHAPTER 3
Readability

This chapter recommends ways of using Ada features to enhance the reader's ability to read and understand code. There are many myths about comments and readability. The responsibility for true readability rests more with naming and code structure than with comments. Having as many comment lines as code lines does not imply readability; it more likely indicates the writer does not understand what is important to communicate.

3.1 SPELLING

Spelling conventions in source code include rules for capitalization, use of underscores, and use of abbreviations. If these conventions are followed consistently, the resulting code is clearer and more readable.

3.1.1 Use of Underscores

guideline

- Use underscores to separate words in a compound name.

example

```
MILES_PER_HOUR
ENTRY_VALUE
```

rationale

When an identifier consists of more than one word, it is much easier to read if the words are separated by underscores. Indeed, there is precedent in English in which compound words are separated by a hyphen. In addition to promoting readability of the code, if underscores are used in names, a code formatter has more control over altering capitalization. See also Guideline 3.1.3.

3.1.2 Numbers

guideline

- Represent numbers in a consistent fashion.
- Represent literals in a radix appropriate to the problem.
- Use underscores to separate digits the same way commas (or spaces for nondecimal bases) would be used in handwritten text.
- When using scientific notation, make the E consistently either upper or lower case.
- In an alternate base, represent the alphabetic characters in either all upper case, or all lower case.

instantiation

- Decimal and octal numbers are grouped by threes beginning counting on either side of the radix point.
- The E is always capitalized in scientific notation.
- Use upper case for the alphabetic characters representing digits in bases above 10.
- Hexadecimal numbers are grouped by fours beginning counting on either side of the radix point.

example

```
type MAXIMUM_SAMPLES     is range  1           .. 1_000_000;
type LEGAL_HEX_ADDRESS   is range 16#0000#     .. 16#FFFF#;
type LEGAL_OCTAL_ADDRESS is range  8#000_000# .. 8#777_777#;

AVOGADRO_NUMBER : constant := 6.022169E+23;
```

To represent the number 1/3 as a constant,

```
-- Use
ONE_THIRD  :  constant  := 1.0/3.0;

-- Avoid this use.
ONE_THIRD  :  constant  := 0.33333333333333;
-- or
ONE_THIRD  :  constant  := 3#0.1#; -- Yes, it really works!
```

rationale

Consistent use of upper case or lower case aids scanning for numbers. Underscores serve to group portions of numbers into familiar patterns. Consistency with common use in everyday contexts is a large part of readability.

note

If a rational fraction is represented in a base in which it has a terminating rather than repeating representation, as `3#0.1#` does in the example above, it may have increased accuracy upon conversion to the machine base.

3.1.3 Capitalization

guideline

- Make reserved words and other elements of the program visually distinct from each other.

instantiation

- Use lower case for all reserved words.
- Use upper case for all other identifiers.

example

```
case TIME_OF_DAY is
   when BEFORE_NOON => GET_READY_FOR_LUNCH;
   when HIGH_NOON   => EAT_LUNCH;
   when AFTER_NOON  => GET_TO_WORK;
end case; -- TIME_OF_DAY
```

rationale

Visually distinguishing reserved words allows the reader to focus on program structure alone if desired and also aids scanning for particular identifiers.

note

(Nissen and Wallis 1984, 2.1) states that "The choice of case is highly debatable, and that chosen for the [Ada Language Reference Manual (Department of Defense 1983)] is not necessarily the best. The use of lower case for reserved words is often preferred, so that they do not stand out too much. However, lower case is generally easier to read than upper case; words can be distinguished by their overall shape, and can be found more quickly when scanning the text."

automation note

Ada names are not case sensitive. Therefore the names `max_limit`, `MAX_LIMIT`, and `Max_Limit` denote the same object or entity. A good code formatter should be able to automatically convert from one style to another, as long as the words are delimited by underscores.

3.1.4 Abbreviations

guideline

- Do not use an abbreviation of a long word as an identifier where a shorter synonym exists.
- Use a consistent abbreviation strategy.
- Do not use uncommon or ambiguous abbreviations.
- An abbreviation must save many characters over the full word to be justified.
- If a project has accepted abbreviations, maintain a list and use only abbreviations on that list.

example

Use `TIME_OF_RECEIPT` rather than `RECD_TIME` or `R_TIME`.

rationale

Many abbreviations are ambiguous or unintelligible unless taken in context. As an example, `TEMP` could indicate either temporary or temperature. For this reason, you should choose abbreviations carefully when you use them.

Since very long variable names can obscure the structure of the program, especially in deeply nested (indented) control structures, it is a good idea to try to keep identifiers short and meaningful. Use short unabbreviated names whenever possible. If there is no short word which will serve as an identifier, then a well known unambiguous abbreviation is the next best choice, especially if it comes from a list of standard abbreviations used throughout the project.

An abbreviated format for a fully qualified name can be established via the renames clause. This capability is useful when a very long fully qualified name would otherwise occur many times in a localized section of code (see Guideline 5.7.2).

A list of accepted abbreviations for a project provides a standard context for the use of each abbreviation.

3.2 NAMING CONVENTIONS

Choose names which make clear the object's or entity's intended use. Ada allows identifiers to be of any length as long as the identifier fits on a line, with all characters being significant. Identifiers are the names used for variables, constants, program units, and other entities within a program.

3.2.1 Names

guideline

- Choose names that are as self-documenting as possible.
- Use a short synonym instead of an abbreviation (see Guideline 3.1.4).
- Use the context to shorten names.
- Reserve the best name for the variable, and the next best for the type.
- Use names given by the application, but not obscure jargon.

example

`TIME_OF_DAY` instead of `TOD`

In a tree-walker, using the name `LEFT` instead of `LEFT_BRANCH` is sufficient to convey the full meaning given the context.

rationale

These attributes can be helpful in comprehending programs. Self-documenting names require fewer explanatory comments. Empirical studies have shown that you can further improve comprehension if your variable names are not excessively long (Schneiderman 1986, 7). The context and application can help greatly. The unit of measure for numeric entities can be a source of type names.

note

The acronyms EDT for Eastern Daylight Time, GMT for Greenwich Mean Time, and FFT for Fast Fourier Transform are good names. They are commonly accepted and widely used and generally are given by the application (but see Guideline 8.1.2). Mathematical formulas are often given using single-letter names for variables. Continue this convention for mathematical equations where it would recall the formula, for example:

```
A * (X**2) + B * X + C.
```

3.2.2 Type Names

guideline

- Choose a name indicative of a category.
- Consider using a plural form as a type name.
- Use specific suffixes.
- If you use suffixes, reserve them only for types.

example

```
type MODE_TYPE is ...       -- too generic
type MODE_NAME is ...       -- specific for an enumeration
type MODE_INFO is ...       -- specific for a record
OPEN_MODE : OPEN_MODES;     -- type name as plural of variable name.
```

rationale

Careful choice of type names clarifies type definitions by conveying meaning about the objects to be declared, and clarifies declarations by indicating the purpose of the declared objects. Using categorical or plural nouns or noun phrases as type names helps to emphasize their generic nature. Suffixes, if used, should be sufficiently specific to convey useful information. Reserving suffixes for type names avoids confusing them with object names which are generic in form.

note

Type names should be descriptive and should preserve the appropriate level of abstraction. For example, EMPLOYEE_LISTS and SETS_OF_EMPLOYEES are better than EMPLOYEE_POINTER_ARRAYS because they describe the data type without revealing the implementation. This is especially important when naming types which are declared in a package specification, as part of an abstract interface. For types declared internally to a unit body, a name like EMPLOYEE_POINTER may be acceptable, especially if the code in the body explicitly manipulates the data type as a pointer.

If there is no good name to describe a category of objects, use a descriptive suffix such as _CLASS, _KIND, etc. (NASA 1987 and United Technologies 1987) However, beware that appending suffixes makes names longer and sometimes awkward. If you use suffixes, use them consistently.

3.2.3 Object Names

guideline

- Use real world object names for objects.
- Use common nouns for nonboolean objects.
- Use predicate clauses or adjectives for boolean objects.
- If you use plural type names, use singular object names.

example

Nonboolean objects:

```
CURRENT_LIST        : LISTS;            -- noun
CLASS_SCHEDULE      : SCHEDULE_TABLES;  -- noun
NUMBER_OF_ELEMENTS  : COUNT;            -- noun phrase
```

Boolean objects:

```
USER_IS_AVAILABLE   : BOOLEAN;          -- predicate clause
LIST_IS_EMPTY       : BOOLEAN;          -- predicate clause
EMPTY               : BOOLEAN;          -- adjective
BRIGHT              : BOOLEAN;          -- adjective
```

rationale

Following conventions which relate object types and parts of speech makes code read more like text. For example, because of the names chosen, the following code segment needs no comments:

```
if LIST_IS_EMPTY then
    NUMBER_OF_ELEMENTS := 0;
else
    NUMBER_OF_ELEMENTS := LENGTH_OF_LIST;
end if;
```

note

If the program is modeling some action in a domain with previously established naming conventions, use the conventions for the domain since they are more familiar to readers of the code.

3.2.4 Program Unit Names

guideline

- Use action verbs for procedures and entries.
- Use predicate-clauses for boolean functions.
- Use nouns for nonboolean functions.
- Give packages names that imply higher levels of organization than subprograms. Generally, these are noun phrases that describe the abstraction provided.
- Give tasks names that imply an active entity.
- Name generic subprograms as if they were nongeneric subprograms.
- Name generic packages as if they were nongeneric packages.
- Make the generic names more general than the instantiated names.

example

The following are sample names for elements that comprise an Ada program.

Sample procedure names:

```
GET_NEXT_TOKEN      -- get is a transitive verb
CREATE_NEW_GROUP    -- create is a transitive verb
```

Sample function names for boolean-valued functions:

```
IS_LAST_ITEM        -- predicate clause
IS_EMPTY            -- predicate clause
```

Sample function names for nonboolean-valued functions:

```
SUCCESSOR           -- common noun
```

Sample package names:

```
TERMINAL_OPERATIONS -- common noun
TEXT_UTILITIES      -- common noun
```

Sample task names:

```
TERMINAL_RESOURCE_MANAGER -- common noun that shows action
```

The following example shows code using the parts-of-speech naming conventions. Below is a sample piece of code to show the clarity that results from using these conventions.

```
GET_NEXT_TOKEN (CURRENT_TOKEN);
case CURRENT_TOKEN is
   when IDENTIFIER => PROCESS_IDENTIFIER;
   when NUMERIC    => PROCESS_NUMERIC;
   ...
end case;  -- CURRENT_TOKEN


if IS_EMPTY (CURRENT_LIST) then
    NUMBER_OF_ELEMENTS := 0;
else
    NUMBER_OF_ELEMENTS := LENGTH_OF (CURRENT_LIST);
end if;
```

When packages and their subprograms are named together, the resulting code is very descriptive.

```
STACK.IS_EMPTY      -- predicate clause
STACK.TOP           -- common noun with prepositional phrase
                    -- used as adjective
SENSOR.READING      -- common noun participle with adjective
```

rationale

Using these naming conventions creates understandable code that reads much like natural language. When verbs are used for actions, such as subprograms, and nouns are used for objects, such as the data that the subprogram manipulates, code is easier to read and understand. This models a medium of communication already familiar to a reader. Where the pieces of a program model a real-life situation, using these conventions reduces the number of translation steps involved in reading and understanding the program. In a sense, your choice of names reflects the level of abstraction from computer hardware toward application requirements.

note

There are some conflicting conventions in current use for task entries. Some programmers and designers advocate naming task entries with the same conventions used for subprograms to blur the fact that a task is involved. Their reasoning is that if the task is reimplemented as a package, or vice versa, the names need not change. Others prefer to make the fact of a task entry as explicit as possible to ensure that the existence of a task with its presumed overhead is recognizable. Project-specific priorities may be useful in choosing between these conventions.

3.2.5 Constants and Named Numbers

guideline

- Use symbolic values instead of literals wherever possible.
- Use constants instead of variables for constant values.
- Use named numbers instead of constants when possible.
- Use named numbers to replace numeric literals whose type or context is truly universal.
- Use constants for objects whose values cannot change after elaboration (Mowday 1986 and United Technologies 1987).
- Show relationships between symbolic values by defining them with static expressions.
- Use linearly independent sets of literals.
- Use attributes like 'SUCC, 'PRED, 'FIRST, and 'LAST instead of literals wherever possible.

example

```
3.141_592_653_589_793                  -- literal
MAX : constant INTEGER := 65_535;      -- constant
PI  : constant         := 3.141_592;   -- named number
PI / 2                                 -- static expression
PI                                     -- symbolic value
```

Declaring PI as a named number allows it to be referenced symbolically in the assignment statement below:

```
AREA := PI * RADIUS**2;            -- if radius is known.
-- instead of
AREA := 3.14159 * RADIUS**2;       -- Needs explanatory comment.
```

Also, ASCII.BEL is more expressive than CHARACTER'VAL(8#007#).

Clarity of constant and named number declarations can be improved by using other constant and named numbers. For example:

```
BYTES_PER_PAGE     : constant := 512;
PAGES_PER_BUFFER   : constant := 10;
BUFFER_SIZE        : constant := PAGES_PER_BUFFER * BYTES_PER_PAGE;

-- is more self-explanatory and easier to maintain than

BUFFER_SIZE : constant := 5120;    -- ten pages
```

The following literals should be constants:

```
if NEW_CHARACTER = '$' -- "constant" that may change
if CURRENT_COLUMN = 7  -- "constant" that may change
```

rationale

Using identifiers instead of literals makes the purpose of expressions clear, reducing the need for comments. Constant declarations consisting of expressions of numeric literals are safer since they need not be computed by hand. They are also more enlightening than a single numeric literal since there is more opportunity for embedding explanatory names. Clarity of constant declarations can be improved further by using other related constants in static expressions defining new constants. This is not less efficient because static expressions of named numbers are computed at compile time.

A constant has a type. A named number can only be of a universal type: universal integer or universal real. Strong typing is enforced for identifiers but not literals. Named numbers allow compilers to generate more efficient code than for constants and to perform more complete error checking at compile time. If the literal contains a large number of digits (as PI in the example above), the use of an identifier reduces keystroke errors. If keystroke errors occur, they are easier to locate either by inspection or at compile time.

Linear independence of literals means that the few literals that are used do not depend on one another and that any relationship between constant or named values is shown in the static expressions. Linear independence of literal values gives the property that if one literal value changes, all of the named numbers of values dependent on that literal are automatically changed.

The literal 1 often occurs in situations where it could be replaced by the 'SUCC and 'PRED attributes. Where these attributes are used instead of the literal 1, the underlying type can be switched more easily during maintenance between numeric and enumeration types. Another benefit of using these attributes is that the operations are more explicit, self-documenting, and instructive than having the maintainer answer such questions as: "If something somewhere else changes, does the 1 change to, say, 5?"

note

There are some gray areas where the literal is actually more self-documenting than a name. These are application-specific and generally occur with universally familiar, unchangeable values such as the following relationship:

```
FAHRENHEIT := 32.0 + (9.0 / 5.0) * CELSIUS;
```

3.3 COMMENTS

Ada comments can be either beneficial or harmful to software maintainers. They can be beneficial by explaining aspects of the code which are otherwise not readily apparent. They can be harmful by containing inaccurate information, and by being too numerous and not visually distinct enough, which can cause them to obscure the structure of the code.

Comments should be minimized. They should explain design decisions, emphasize the structure of code, and draw attention to deliberate and necessary violations of the guidelines. It is important to note that many of the examples in this book include more comments than we would generally consider necessary or advisable. They are present either to draw attention to the real issue that is being exemplified or to compensate for incompleteness in the example program.

Maintenance programmers need to know the causal interaction of noncontiguous pieces of code to get a global, more or less complete sense of the program. They typically acquire this kind of information from mental simulation of parts of the code. Comments should be just sufficient to support this process (Soloway et al. 1986).

This section presents general guidelines about how to write good comments, and then defines several different classes of comments with guidelines for the use of each. The classes are: file headers, program unit specification headers, program unit body headers, data comments, statement comments, and marker comments.

3.3.1 General Comments

guideline

- Make the code as clear as possible to reduce the need for comments.
- Never repeat information in a comment which is readily available in the code.
- Where a comment is required, make it concise and complete.
- Use proper grammar and spelling in comments.
- Make comments visually distinct from the code.
- Comments in headers should be structured so that information can be extracted from them automatically by a tool.

rationale

The structure and function of well written code is clear without comments. Obscure or badly structured code is hard to understand, maintain, or reuse regardless of comments. Bad code should be improved, not explained. Reading the code itself is the only way to be absolutely positive about what the code does. Therefore, the code should be made as readable as possible.

Using comments to duplicate information in the code is a bad idea for several reasons. First, it is unnecessary work which decreases productivity. Second, it is very difficult to maintain the duplication correctly as the code is modified. When changes are made to existing code, it is compiled and tested to make sure that it is once again correct. However, there is no automatic mechanism to make sure that the comments are updated correctly to reflect the changes. Very often the duplicate information in a comment becomes obsolete at the first code change, and remains so through the life of the software.

Third, when comments about an entire system are written from the limited point of view of the author of a single subsystem, they are often incorrect from the start.

Comments are necessary to reveal information which is difficult or impossible to obtain from the code. Subsequent sections of this book contain examples of such comments. Completely and concisely present the required information.

The purpose of comments is to help readers understand the code. Misspelled, ungrammatical, ambiguous, or incomplete comments defeat this purpose. If a comment is worth adding, it is worth adding correctly in order to increase its usefulness.

Making comments visually distinct from the code, by indenting them, grouping them together into headers, or highlighting them with dashed lines is useful because it makes the code easier to read. Subsequent sections of this book elaborate on this point.

automation note

The guideline about storing redundant information in comments applies only to manually generated comments. There are tools which automatically maintain information about the code (e.g., calling units, called units, cross-reference information, revision histories, etc.), storing it in comments in the same file as the code. Other tools read comments, but do not update them, using the information from the comments to automatically generate detailed design documents and other reports.

The use of such tools is encouraged, and may require that you structure your header comments so they can be automatically extracted and/or updated. Beware that tools which modify the comments in a file are only useful if they are executed frequently enough. Automatically generated obsolete information is even more dangerous than manually generated obsolete information, because it is more trusted by the reader.

Revision histories are maintained much more accurately and completely by configuration management tools. With no tool support, it is very common for an engineer to make a change and forget to update the revision history. If your configuration management tool is capable of maintaining revision histories as comments in the source file, then take advantage of that capability, regardless of any compromise you might have to make about the format or location of the revision history. It is better to have a complete revision history appended to the end of the file than to have a partial one formatted nicely and embedded in the file header.

3.3.2 File Headers

guideline

- Put a file header on each source file.
- Place ownership, responsibility, and history information for the file in the file header.

instantiation

- Put a copyright notice in the file header.
- Put the author's name and department in the file header.
- Put a revision history in the file header, including a summary of each change, the date, and the name of the person making the change.

example

```
------------------------------------------------------------------------
--
--      Copyright (c) 1991, Software Productivity Consortium, Inc.
--      All rights reserved.
--
--
-- Author:       J. Smith
--
-- Department:   System Software Department
--
-- Revision History:
--   7/9/91      J. Smith
--     - Added function SIZE_OF to support queries of node sizes.
--     - Fixed bug in SET_SIZE which caused overlap of large nodes.
--   7/1/91      M. Jones
--     - Optimized clipping algorithm for speed.
--   6/25/91     J. Smith
--     - Original version.
------------------------------------------------------------------------
```

rationale

Ownership information should be present in each file if you want to be sure to protect your rights to the software. Furthermore, for high visibility, it should be the very first thing in the file.

Responsibility and revision history information should be present in each file for the sake of future maintainers, this is the header information most trusted by maintainers because it accumulates. It does not evolve. There is no need to ever go back and modify the author's name or the revision history of a file. As the code evolves, the revision history should be updated to reflect each change. At worst, it will be incomplete, it should rarely be wrong. Also, the number and frequency of changes and the number of different people who made the changes over the history of a unit can be good indicators of the integrity of the implementation with respect to the design.

Information about how to find the original author should be included in the file header, in addition to the author's name, to make it easier for maintainers to find the author in case questions arise. However, detailed information like phone numbers, mail stops, office numbers, and computer account usernames are too volatile to be very useful. It is better to record the department for which the author was working when the code was written. This information is still useful if the author moves offices, changes departments, or even leaves the company, because the department is likely to retain responsibility for the original version of the code.

3.3.3 Program Unit Specification Header

guideline

- Put a header on the specification of each program unit.
- Place information required by the user of the program unit in the specification header.
- Do not repeat information (except unit name) in the specification header which is present in the specification.
- Explain what the unit does, not how or why it does it.
- Describe the complete interface to the program unit, including any exceptions it can raise and any global effects it can have.
- Do not include information about how the unit fits into the enclosing software system.
- Describe the performance (time and space) characteristics of the unit.

instantiation

- Put the name of the program unit in the header.
- Briefly explain the purpose of the program unit.
- For packages, describe the effects of the visible subprograms on each other, and how they should be used together.

- List all exceptions which can be raised by the unit.
- List all global effects of the unit.
- List preconditions and postconditions of the unit.
- List hidden tasks activated by the unit.
- Do not list the names of parameters of a subprogram.
- Do not list the names of subprograms of a package.
- Do not list the names of all other units used by the unit.
- Do not list the names of all other units which use the unit.

example

```
-------------------------------------------------------------------------
-- AUTOLAYOUT
-------------------------------------------------------------------------
-- Purpose:
--   This package computes positional information for nodes and arcs of a
--   directed graph.  It encapsulates a layout algorithm which is designed
--   to minimize the number of crossing arcs and to emphasize the primary
--   direction of arc flow through the graph.
--
-- Effects:
--   - The ex[illegible]d usage is:
--     1. Call DEFINE for each node and arc to define the graph.
--     2. [illegible] LAYOUT to assign positions to all nodes and arcs.
--     3. Call POSITION_OF for each node and arc to determine the
--        assigned coordinate positions.
--   - LAYOUT can be called multiple times, and recomputes the
--     positions of all currently defined nodes and arcs each time.
--   - Once a node or arc has been defined, it remains defined until
--     CLEAR is called to delete all nodes and arcs.
--
-- Performance:
--   This package has been optimized for time, in preference to space.
--   Layout times are on the order of N*log(N) where N is the number of
--   nodes, but memory space is used inefficiently.
-------------------------------------------------------------------------
package AUTOLAYOUT is

   ...

   -------------------------------------------------------------------------
   -- DEFINE
   -------------------------------------------------------------------------
   -- Purpose:
   --   This procedure defines one node of the current graph.
   -- Exceptions:
   --   NODE_ALREADY_DEFINED
   -------------------------------------------------------------------------
   procedure DEFINE (NODE : NODE_TYPE);

   -------------------------------------------------------------------------
   -- LAYOUT
   -------------------------------------------------------------------------
   -- Purpose:
   --   This procedure assigns coordinate positions to all defined nodes
   --   and arcs.
   -- Exceptions:
   --   None.
   -------------------------------------------------------------------------
   procedure LAYOUT;
```

```
        ------------------------------------------------------------------------
        -- POSITION_OF
        ------------------------------------------------------------------------
        -- Purpose:
        --    This function returns the coordinate position of the specified
        --    node.  The default position (0,0) is returned if no position has
        --    been assigned yet.
        -- Exceptions:
        --    NODE_NOT_DEFINED
        ------------------------------------------------------------------------
        function POSITION_OF (NODE : NODE_TYPE) return POSITION;

        ...

    end AUTOLAYOUT;
```

rationale

The purpose of a header comment on the specification of a program unit is to help the user understand how to use the program unit. From reading the program unit specification and header, a user should know everything necessary to use the unit. It should not be necessary to read the body of the program unit. Therefore, there should be a header comment on each program unit specification, and each header should contain all usage information which is not expressed in the specification itself. Such information includes effects of units on each other and on shared resources, exceptions raised, and time/space characteristics of units. None of this information can be determined from the Ada specification of the program unit.

When you duplicate information in the header that can be readily obtained from the specification, the information tends to become incorrect during maintenance. For example, do not make a point of listing all parameter names, modes or types when describing a procedure. This information is already available from the procedure specification. Similarly, do not list all subprograms of a package in the header, unless this is necessary to make some important statement about the subprograms.

Do not include information in the header which the user of the program unit doesn't need. In particular, do not include information about how a program unit performs its function, or why a particular algorithm was used. This information should be hidden in the body of the program unit, to preserve the abstraction defined by the unit. If the user knows such details and makes decisions based on that information, the code may suffer when that information is later changed.

When describing the purpose of the unit, avoid referring to other parts of the enclosing software system. It is better to say "this unit does ...", than to say "this unit is called by XYZ to do ...". The unit should be written in such a way that it does not know or care which unit is calling it. This makes the unit much more general purpose and reusable. In addition, information about other units is likely to become obsolete and incorrect during maintenance.

Include information about the performance (time and space) characteristics of the unit. Much of this information is not present in the Ada specification, but is required by the user. To integrate the unit into a system, the user needs to understand the resource usage (CPU, memory, etc.) of the unit. It is especially important to note when a subprogram call causes activation of a task that is hidden in a package body because this task may continue to consume resources after the end of the subroutine, and similarly for tasks activated by "`with`"ing a package.

exception

Where a group of program units are closely related or simple to understand, it is acceptable to use a single header for the entire group of program units. For example, it makes sense to use a single header to describe the behavior of MAX and MIN functions, or SIN, COS, and TAN functions, or a group of functions to query related attributes of an object encapsulated in a package. This is especially true when each function in the set is capable of raising the same exceptions.

3.3.4 Program Unit Body Header

guideline

- Place information required by the maintainer of the program unit in the body header.
- Explain how and why the unit performs its function, not what the unit does.

- Do not repeat information (except unit name) in the header which is readily apparent from reading the code.
- Do not repeat information (except unit name) in the body header which is available in the specification header.

instantiation

- Put the name of the program unit in the header.
- Record portability issues in the header.
- Summarize complex algorithms in the header.
- Record reasons for significant or controversial implementation decisions.
- Record discarded implementation alternatives, along with the reason for discarding them.
- Record anticipated changes in the header, especially if some work has already been done to the code to make the changes easy to accomplish.

example

```
-------------------------------------------------------------------------
-- AUTOLAYOUT
-------------------------------------------------------------------------
-- Implementation Notes:
--   - This package uses a heuristic algorithm to minimize the number of
--     arc crossings.  It does not always achieve the true minimum number
--     which could theoretically be reached.  However it does a nearly
--     perfect job in relatively little time.  For details about the
--     algorithm, see ...
-- Portability Issues:
--   - The native math package MATH_LIB is used for computations of
--     coordinate positions.
--   - 32-bit integers are required.
--   - No operating system specific routines are called.
-- Anticipated Changes:
--   - COORDINATE_TYPE below could be changed from integer to float with
--     little effort.  Care has been taken to not depend on the specific
--     characteristics of integer arithmetic.
-------------------------------------------------------------------------
package body AUTOLAYOUT is

   ...

   -------- ---------- ---------------------------------------------------
   -- DEFINE
   -------------------------------------------------------------------------
   -- Implementation Notes:
   --   - This routine stores a node in the general purpose GRAPH data
   --     structure, not the FAST_GRAPH structure because ...
   -------------------------------------------------------------------------
   procedure DEFINE (NODE : NODE_TYPE) is
   begin
      ...
   end DEFINE;

   -------------------------------------------------------------------------
   -- LAYOUT
   -------------------------------------------------------------------------
   -- Implementation Notes:
   --   - This routine copies the GRAPH data structure (optimized for fast
   --     random access) into the FAST_GRAPH data structure (optimized for
   --     fast sequential iteration), then performs the layout, and copies
   --     the data back to the GRAPH structure.  This technique was
   --     introduced as an optimization when the algorithm was found to be
   --     too slow, and it produced an order of magnitude improvement.
   -------------------------------------------------------------------------
   procedure LAYOUT is
   begin
      ...
   end LAYOUT;
```

```
   ------------------------------------------------------------------------
   -- POSITION_OF
   ------------------------------------------------------------------------
   function POSITION_OF (NODE : NODE_TYPE) return POSITION is
   begin
      ...
   end POSITION_OF;

   ...

end AUTOLAYOUT;
```

rationale

The purpose of a header comment on the body of a program unit is to help the maintainer of the program unit to understand the implementation of the unit, including tradeoffs among different techniques. Be sure to document all decisions made during implementation to prevent the maintainer from making the same mistakes you made. One of the most valuable comments to a maintainer is a clear description of why a change being considered will not work.

The header is also a good place to record portability concerns. The maintainer may have to port the software to a different environment and will benefit from a list of nonportable features of which the author was aware. Furthermore, the act of collecting and recording portability issues focuses the author's attention on these issues and may result in more portable code from the start.

Summarize complex algorithms in the header if the code is difficult to read or understand without such a summary, but do not merely paraphrase the code. Such duplication is unnecessary and hard to maintain. Similarly, do not repeat the information from the header of the program unit specification.

note

It is often the case that a program unit is self-explanatory enough that it requires no body header to explain how it is implemented or why. In such a case, omit the header entirely, as in the case with POSITION_OF above. Be sure, however, that the header you omit truly contains no information. For example, consider the difference between the two header sections:

```
-- Implementation Notes:  None.
```

and

```
-- Non-Portable Features:  None.
```

The first is a message from the author to the maintainer saying "I can't think of anything else to tell you" while the second may mean "I guarantee that this unit is entirely portable."

3.3.5 Data Comments

guideline

- Comment all data types, objects, and exceptions unless their names are completely self-explanatory.
- Include information on the semantic structure of complex pointer-based data structures.
- Include information about relationships which are maintained between data objects.
- Do not include comments which merely repeat the information in the name.

example

Objects can be grouped by purpose and commented as:

```
-------------------------------------------------------------------------
-- Current position of the cursor in the currently selected text buffer,
-- and the most recent position explicitly marked by the user.
--
-- Note:  It is necessary to maintain both current and desired column
--        positions because the cursor cannot always be displayed in
--        the desired position when moving between lines of different
--        lengths.
-------------------------------------------------------------------------
DESIRED_COLUMN : COLUMN_COUNTERS;
CURRENT_COLUMN : COLUMN_COUNTERS;
CURRENT_ROW    : ROW_COUNTERS;
MARKED_COLUMN  : COLUMN_COUNTERS;
MARKED_ROW     : ROW_COUNTERS;
```

The conditions under which an exception is raised should be commented:

```
-------------------------------------------------------------------------
-- Exceptions
-------------------------------------------------------------------------
NODE_ALREADY_DEFINED : exception;     -- Raised when an attempt is made to
                                      --  define a node with an identifier
                                      --  which already defines a node.
NODE_NOT_DEFINED     : exception;     -- Raised when a reference is made
                                      --  to a node which has not been
                                      --  defined.
```

Here is a more complex example, involving multiple record and access types which are used to form a complex data structure:

```
-------------------------------------------------------------------------
-- These data structures are used to store the graph during the layout
-- process. The overall organization is a sorted list of "ranks," each
-- containing a sorted list of nodes, each containing a list of incoming
-- arcs and a list of outgoing arcs.
--
-- The lists are doubly linked to support forward and backward passes
-- for sorting. Arc lists do not need to be doubly linked because
-- order of arcs is irrelevant.
--
-- The nodes and arcs are doubly linked to each other to support efficient
-- lookup of all arcs to/from a node, as well as efficient lookup of the
-- source/target node of an arc.
-------------------------------------------------------------------------
```

```
type ARCS;
type ARC_PTRS is access ARCS;
type ARCS is
   record
      ID     : ARC_TYPE;        -- Unique arc ID supplied by the user.
      SOURCE : NODE_PTRS;
      TARGET : NODE_PTRS;
      NEXT   : ARC_PTRS;
   end record;

type NODES;
type NODE_PTRS is access NODES;
type NODES is
   record
      ID       : NODE_TYPE;     -- Unique node ID supplied by the user.
      ARCS_IN  : ARC_PTRS;
      ARCS_OUT : ARC_PTRS;
      NEXT     : NODE_PTRS;
      PREVIOUS : NODE_PTRS;
   end record;

type RANKS;
type RANK_PTRS is access RANKS;
type RANKS is
   record
      NUMBER     : LEVEL_TYPE;  -- Computed ordinal number of the rank.
      FIRST_NODE : NODE_PTRS;
      LAST_NODE  : NODE_PTRS;
      NEXT       : RANK_PTRS;
      PREVIOUS   : RANK_PTRS;
   end record;

FIRST_RANK : RANK_PTRS;LAST_RANK  : RANK_PTRS;
```

rationale

It is very useful to add comments explaining the purpose, structure, and semantics of the data structures. Many maintainers look at the data structures first when trying to understand the implementation of a unit. Understanding the data which can be stored, along with the relationships between the different data items, and the flow of data through the unit is an important first step in understanding the details of the unit.

In the first example above, the names CURRENT_COLUMN and CURRENT_ROW are relatively self-explanatory. The name DESIRED_COLUMN is also well-chosen, but leaves the reader wondering what the relationship is between the current column and the desired column. The comment explains the reason for having both.

Another advantage of commenting the data declarations is that the single set of comments on a declaration can replace multiple sets of comments which might otherwise be needed at various places in the code where the data is manipulated. In the first example above, the comment briefly expands on the meaning of "current" and "marked," stating that the "current" position is the location of the cursor, the "current" position is in the current buffer, and the "marked" position was marked by the user. This comment, along with the mnemonic names of the variables, greatly reduces the need for comments at individual statements throughout the code:

It is important to document the full meaning of exceptions and under what conditions they can be raised, as shown in the second example above, especially when the exceptions are declared in a package specification. The reader has no other way to find out the exact meaning of the exception (without reading the code in the package body).

Grouping all the exceptions together, as shown in the second example above, can provide the reader with the effect of a "glossary" of special conditions. This is useful when many different subprograms in the package can raise the same exceptions. For a package in which each exception can be raised by only one subprogram, it may be better to group related subprograms and exceptions together.

When commenting exceptions, it is better to describe the meaning of the exception in general terms than to list all the subprograms that can cause the exception to be raised, such a list is harder to maintain. When a new routine is added in the future, it is likely that these lists will not be updated. Also, this information is already present in the comments describing the subprograms, where all exceptions

that can be raised by the subprogram should be listed. Lists of exceptions by subprogram are more useful and easier to maintain than lists of subprograms by exception.

In the third example above, the names of the record fields are chosen to be short and mnemonic, but they are not completely self-explanatory. This is often the case with complex data structures involving access types. There is no way to choose the record and field names so that they completely explain the overall organization of the records and pointers into a nested set of sorted lists. The comments shown are useful in this case. Without them, the reader would not know which lists are sorted, which lists are doubly linked, or why. The comments express the intent of the author with respect to this complex data structure. The maintainer still has to read the code if he wants to be sure that the double links are all properly maintained. Keeping this in mind when he reads the code makes it much easier for him to find a bug where one pointer is updated and the opposite one is not.

3.3.6 Statement Comments

guideline

- Minimize comments embedded among statements.
- Use comments only to explain parts of the code which are not obvious.
- Comment intentional omissions from the code.
- Do not use comments to paraphrase the code.
- Do not use comments to explain remote pieces of code, such as subprograms called by the current unit.
- Where comments are necessary, make them visually distinct from the code.

example

The following is an example of very poorly commented code:

```
-- Loop through all the strings in the array STRINGS, converting them to
-- integers by calling CONVERT_TO_INTEGER on each one, accumulating the
-- sum of all the values in SUM, and counting them in COUNT.  Then
-- divide SUM by COUNT to get the average and store it in AVERAGE.
-- Also, record the maximum number in the global variable MAX_NUMBER.
for I in STRINGS'range loop
   -- Convert each string to an integer value by looping through the
   -- characters which are digits, until a non-digit is found, taking
   -- the ordinal value of each, subtracting the ordinal value of '0',
   -- and multiplying by 10 if another digit follows.  Store the result
   -- in NUMBER.
   NUMBER := CONVERT_TO_INTEGER (STRINGS (I));
   -- Accumulate the sum of the numbers in TOTAL.
   SUM := SUM + NUMBER;
   -- Count the numbers.
   COUNT := COUNT + 1;
   -- Decide whether this number is more than the current maximum.
   if NUMBER > MAX_NUMBER then
      -- Update the global variable MAX_NUMBER.
      MAX_NUMBER := NUMBER;
   end if;
end loop;
-- Compute the average.
AVERAGE := SUM / COUNT;
```

The following is improved by not repeating things in the comments which are obvious from the code, not describing the details of what goes in inside of CONVERT_TO_INTEGER, deleting an erroneous comment (the

one on the statement which accumulates the sum), and making the few remaining comments more visually distinct from the code.

```
-------------------------------------------------------------------------
-- Compute the average.
-------------------------------------------------------------------------
for I in STRINGS'range loop
   NUMBER := CONVERT_TO_INTEGER (STRINGS (I));
   SUM    := SUM    + NUMBER;
   COUNT  := COUNT + 1;
   if NUMBER > MAX_NUMBER then
      MAX_NUMBER := NUMBER;
                          -- Note:  The global MAX_NUMBER is computed
                          --        here for efficiency.
   end if;
end loop;

AVERAGE := SUM / COUNT;
```

rationale

The improvements shown in the example above are not improvements merely by reducing the number of comments; they are improvements by reducing the number of useless comments.

Comments which paraphrase the code, or explain obvious aspects of the code have no value. They are a waste of effort for the author to write and the maintainer to update. Therefore, they often end up becoming incorrect. Such comments also clutter the code, hiding the few important comments.

Comments which describe what goes on inside of another unit violate the principle of information hiding. The details about CONVERT_TO_INTEGER deleted above are irrelevant to the calling unit, and are better left hidden in case the algorithm ever changes. Examples which explain what goes on elsewhere in the code are very difficult to maintain, and almost always become incorrect at the first code modification.

The advantage of making comments visually distinct from the code is that it makes the code easier to scan, and the few important comments stand out better. Highlighting unusual or special code features indicates that they are intentional. This assists maintainers by focusing attention on code sections that are likely to cause problems during maintenance or when porting the program to another implementation.

Comments should be used to document code that is nonportable, implementation-dependent, environment-dependent, or tricky in any way. They notify the reader that something unusual was put there for a reason. A beneficial comment would be one explaining a work-around for a compiler bug. If you use a lower level (not "ideal" in the software engineering sense) solution, comment it. Information included in the comments should state why you used that particular construct. Also include documentation of the failed attempts, e.g., using a higher level structure. This type of comment is useful to maintainers for historical purposes. Show the reader that a significant amount of thought went into the choice of a construct.

Finally, comments should be used to explain what is not present in the code, as well as what is present. If you make a conscious decision to not perform some action, like deallocating a data structure with which you appear to be finished, be sure to add a comment explaining why not. Otherwise, a maintainer may notice the apparent omission and "correct" it later, introducing an error.

note

Further improvements could be made on the above example by declaring the variables COUNT and SUM in a local block so that their scope is limited and their initializations occur near their usage. For example, by naming the block COMPUTE_AVERAGE or by moving the code into a function called AVERAGE_OF. The computation of MAX_NUMBER could also be separated from the computation of AVERAGE. However, those changes are the subject of other guidelines; this example is intended only to illustrate the proper use of comments.

3.3.7 Marker Comments

guideline

- Use pagination markers to mark program unit boundaries (Guideline 2.1.7).

- Repeat the unit name in a comment to mark the `begin` of a package body, subprogram body, task body, or block if the `begin` is preceded by declarations.
- For long or heavily nested `if` and `case` statements, mark the end of the statement with a comment summarizing the condition governing the statement.
- For long or heavily nested `if` statements, mark the `else` and `elsif` parts with a comment summarizing the conditions governing this portion of the statement.

example

```
if A_FOUND then
   ...
   ...
   ...
elsif B_FOUND then -- A was not found
   ...
   ...
   ...
else -- A and B were both not found
   ...
   if COUNT = MAX then
      ...
   end if;   ...
   ...
end if; -- A_FOUND

------------------------------------------------------------------------
package body ABSTRACT_STRINGS is
   ...
   ------------------------------------------------------------------------
   procedure CATENATE( ... ) is
      ...
   end CATENATE;
   ------------------------------------------------------------------------
   ...
   ...
   ...
   ------------------------------------------------------------------------
begin -- ABSTRACT_STRINGS
   ...
end ABSTRACT_STRINGS;
------------------------------------------------------------------------
```

rationale

Marker comments emphasize the structure of code and make it easier to scan. They can be lines that separate sections of code or descriptive tags for a construct. They help the reader in resolving questions about the current position in the code. This is more important for large units than for small ones. A short marker comment fits on the same line as the reserved word with which it is associated. Thus, it adds information without clutter.

The `if`, `elsif`, `else`, and `end if` of an if statement are often separated by long sequences of statements, sometimes involving other if statements. As shown in the first example above, marker comments emphasize the association of the keywords of the same statement over a great visual distance. Marker comments are not necessary with the block statement and loop statement because the syntax of these statements allows them to be named, with the name repeated at the end. Using these names is better than using marker comments because the compiler verifies that the names at the beginning and end match.

The sequence of statements of a package body is often very far from the first line of the package. Many subprogram bodies each containing many `begin` lines may occur first. As shown in the second example above, the marker comment emphasizes the association of the `begin` with the package.

note

Repeating names and noting conditional expressions clutters the code if overdone. It is visual distance, especially page breaks, that makes marker comments beneficial.

3.4 USING TYPES

Strong typing promotes reliability in software. The type definition of an object defines all legal values and operations and allows the compiler to check for and identify potential errors during compilation. In addition, the rules of type allow the compiler to generate code to check for violations of type constraints at execution time. Using these Ada compilers features facilitates earlier and more complete error detection than that which is available with less strongly typed languages.

3.4.1 Declaring Types

guideline

- Limit the range of scalar types as much as possible.
- Seek information about possible values from the application.
- Do not overload any of the type names in package STANDARD.
- Use subtype declarations to improve program readability (Booch 1987).
- Use derived types and subtypes in concert (see Guideline 5.3.1).

example

```
subtype CARD_IMAGE is STRING (1 .. 80);
INPUT_LINE : CARD_IMAGE := (others => ' ');

-- restricted integer type:
type DAY_OF_LEAP_YEAR is range 1 .. 366;
subtype DAY_OF_NON_LEAP_YEAR is DAY_OF_LEAP_YEAR range 1 .. 365;
```

By the following declaration, the programmer means, "I haven't the foggiest idea how many," but the actual range will show up buried in the code or as a system parameter:

```
EMPLOYEE_COUNT : INTEGER;
```

rationale

Eliminating meaningless values from the legal range improves the compiler's ability to detect errors when an object is set to an invalid value. This also improves program readability. In addition, it forces you to think carefully about each use of objects declared to be of the subtype.

Different implementations provide different sets of values for most of the predefined types. A reader cannot determine the intended range from the predefined names. This situation is aggravated when the predefined names are overloaded.

The names of an object and its subtype can make clear their intended use and document low-level design decisions. The example above documents a design decision to restrict the software to devices whose physical parameters are derived from the characteristics of punch cards. This information is easy to find for any later changes, enhancing program maintainability.

Declaration of a subtype without a constraint is one method for renaming a type [Ada Language Reference Manual (Department of Defense 1983, §8.5)].

Types can have highly constrained sets of values without eliminating useful values. Usage as described in Guideline 5.3.1 eliminates many flag variables and type conversions within executable statements. This renders the program more readable while allowing the compiler to enforce strong typing constraints.

note

Subtype declarations do not define new types, only constraints for existing types.

Recognize that any deviation from this guideline detracts from the advantages of the strong typing facilities of the Ada language.

3.4.2 Enumeration Types

guideline

- Use enumeration types instead of numeric codes.
- Use representation clauses to match requirements of external devices.

example

```
-- Use

type COLORS is
  (BLUE,
   RED,
   GREEN,
   YELLOW);

-- rather than

BLUE   : constant := 1;
RED    : constant := 2;
GREEN  : constant := 3;
YELLOW : constant := 4;

-- and add the following if necessary.

for COLORS use
  (BLUE   => 1,
   RED    => 2,
   GREEN  => 3,
   YELLOW => 4);
```

rationale

Enumerations are more robust than numeric codes; they leave less potential for errors resulting from incorrect interpretation, and from additions to and deletions from the set of values during maintenance. Numeric codes are holdovers from languages that have no user-defined types.

In addition, Ada provides a number of attributes (`'POS`, `'VAL`, `'SUCC`, `'PRED`, `'IMAGE`, and `'VALUE`) for enumeration types which, when used, are more reliable than user-written operations on encodings.

A numeric code might at first seem appropriate to be certain that specific values match requirements for signals on control lines or expected inputs from sensors. These situations instead call for a representation clause on the enumeration type. The representation clause documents the "*encoding.*" If the program is properly structured to isolate and encapsulate hardware dependencies (see Guideline 7.1.5), the numeric code ends up in an interface package where it can be easily found and replaced, should the requirements change.

3.5 SUMMARY

spelling

- Use underscores to separate words in a compound name.
- Represent numbers in a consistent fashion.
- Represent literals in a radix appropriate to the problem.
- Use underscores to separate digits the same way commas (or spaces for nondecimal bases) would be used in handwritten text.
- When using scientific notation, make the `E` consistently either upper or lower case.
- In an alternate base, represent the alphabetic characters in either all upper case, or all lower case.
- Make reserved words and other elements of the program visually distinct from each other.
- Do not use an abbreviation of a long word as an identifier where a shorter synonym exists.
- Use a consistent abbreviation strategy.
- Do not use uncommon or ambiguous abbreviations.
- An abbreviation must save many characters over the full word to be justified.
- If a project has accepted abbreviations, maintain a list and use only abbreviations on that list.

naming conventions

- Choose names that are as self-documenting as possible.

- Use a short synonym instead of an abbreviation.
- Use the context to shorten names.
- Reserve the best name for the variable, and the next best for the type.
- Use names given by the application, but not obscure jargon.
- Choose a name indicative of a category.
- Consider using a plural form as a type name.
- Use specific suffixes.
- If you use suffixes, reserve them only for types.
- Use real world object names for objects.
- Use common nouns for nonboolean objects.
- Use predicate clauses or adjectives for boolean objects.
- If you use plural type names, use singular object names.
- Use action verbs for procedures and entries.
- Use predicate-clauses for boolean functions.
- Use nouns for nonboolean functions.
- Give packages names that imply higher levels of organization than subprograms. Generally, these are noun phrases that describe the abstraction provided.
- Give tasks names that imply an active entity.
- Name generic subprograms as if they were nongeneric subprograms.
- Name generic packages as if they were nongeneric packages.
- Make the generic names more general than the instantiated names.
- Use symbolic values instead of literals wherever possible.
- Use constants instead of variables for constant values.
- Use named numbers instead of constants when possible.
- Use named numbers to replace numeric literals whose type or context is truly universal.
- Use constants for objects whose values cannot change after elaboration.
- Show relationships between symbolic values by defining them with static expressions.
- Use linearly independent sets of literals.
- Use attributes like `'SUCC`, `'PRED`, `'FIRST`, and `'LAST` instead of literals wherever possible.

comments

- Make the code as clear as possible to reduce the need for comments.
- Never repeat information in a comment which is readily available in the code.
- Where a comment is required, make it concise and complete.
- Use proper grammar and spelling in comments.
- Make comments visually distinct from the code.
- Comments in headers should be structured so that information can be extracted from them automatically by a tool.
- Put a file header on each source file.
- Place ownership, responsibility, and history information for the file in the file header.
- Put a header on the specification of each program unit.
- Place information required by the user of the program unit in the specification header.

- Do not repeat information (except unit name) in the specification header which is present in the specification.
- Explain what the unit does, not how or why it does it.
- Describe the complete interface to the program unit, including any exceptions it can raise and any global effects it can have.
- Do not include information about how the unit fits into the enclosing software system.
- Describe the performance (time and space) characteristics of the unit.
- Place information required by the maintainer of the program unit in the body header.
- Explain how and why the unit performs its function, not what the unit does.
- Do not repeat information (except unit name) in the header which is readily apparent from reading the code.
- Do not repeat information (except unit name) in the body header which is available in the specification header.
- Comment all data types, objects, and exceptions unless their names are completely self-explanatory.
- Include information on the semantic structure of complex pointer-based data structures.
- Include information about relationships which are maintained between data objects.
- Do not include comments which merely repeat the information in the name.
- Minimize comments embedded among statements.
- Use comments only to explain parts of the code which are not obvious.
- Comment intentional omissions from the code.
- Do not use comments to paraphrase the code.
- Do not use comments to explain remote pieces of code, such as subprograms called by the current unit.
- Where comments are necessary, make them visually distinct from the code.
- Use pagination markers to mark program unit boundaries.
- Repeat the unit name in a comment to mark the `begin` of a package body, subprogram body, task body, or block if the `begin` is preceded by declarations.
- For long or heavily nested `if` and `case` statements, mark the end of the statement with a comment summarizing the condition governing the statement.
- For long or heavily nested `if` statements, mark the `else` and `elsif` parts with a comment summarizing the conditions governing this portion of the statement.

using types

- Limit the range of scalar types as much as possible.
- Seek information about possible values from the application.
- Do not overload any of the type names in package STANDARD.
- Use subtype declarations to improve program readability.
- Use derived types and subtypes in concert.
- Use enumeration types instead of numeric codes.
- Use representation clauses to match requirements of external devices.

CHAPTER 4
Program Structure

Proper structure improves program clarity. This is analogous to readability on lower levels and facilitates the use of the readability guidelines (Chapter 3). The various program structuring facilities provided by Ada were designed to enhance overall clarity of design. These guidelines show how to use these facilities for their intended purposes.

Abstraction and encapsulation are supported by the package concept and by private types. Related data and subprograms can be grouped together and seen by a higher level as a single entity. Information hiding is enforced via strong typing and by the separation of package and subprogram specifications from their bodies. Additional Ada language elements that impact program structure are exceptions and tasks.

4.1 *HIGH-LEVEL STRUCTURE*

Well-structured programs are easily understood, enhanced, and maintained. Poorly structured programs are frequently restructured during maintenance just to make the job easier. Many of the guidelines listed below are often given as general program design guidelines.

4.1.1 Separate Compilation Capabilities

guideline

- Place the specification of each library unit package in a separate file from its body.
- Create an explicit specification, in a separate file, for each library unit subprogram.
- Use subunits for the bodies of large units which are nested in other units.
- Place each subunit in a separate file.
- Use a consistent file naming convention.

example

The file names below illustrate one possible file organization and associated consistent naming convention. The library unit name is used for the body. A trailing underscore indicates the specification, and any files containing subunits use names constructed by separating the body name from the subunit name with two underscores.

```
TEXT_IO_.ADA                -- the specification
TEXT_IO.ADA                 -- the body
TEXT_IO__INTEGER_IO.ADA     -- a subunit
TEXT_IO__FIXED_IO.ADA       -- a subunit
TEXT_IO__FLOAT_IO.ADA       -- a subunit
TEXT_IO__ENUMERATION_IO.ADA -- a subunit
```

rationale

The main reason for the emphasis on separate files in this guideline is to minimize the amount of recompilation required after each change. Typically, during software development, bodies of units are

updated far more often than specifications. If the body and specification reside in the same file, then the specification will be compiled each time the body is compiled, even though the specification has not changed. Because the specification defines the interface between the unit and all of its users, this recompilation of the specification typically makes recompilation of all users necessary, in order to verify compliance with the specification. If the specifications and bodies of the users also reside together, then any users of these units will also have to be recompiled, and so on. The ripple effect can force a huge number of compilations which could have been avoided, severely slowing the development and test phase of a project. This is why we suggest placing specifications of all library units (nonnested units) in separate files from their bodies.

For the same reason, use subunits for large nested bodies, and put each subunit in its own file. This makes it possible to modify the body of the one nested unit without having to recompile any of the other units in the body. This is recommended for large units because changes are more likely to occur in large units than in small ones.

An additional benefit of using multiple separate files is that it allows different implementers to modify different parts of the system at the same time with conventional editors which do not allow multiple concurrent updates to a single file.

Finally, keeping bodies and specifications separate makes it possible to have multiple bodies for the same specification, or multiple specifications for the same body. Although Ada requires that there be exactly one specification per body in a system at any given time, it can still be useful to maintain multiple bodies or multiple specifications for use in different builds of a system. For example, a single specification may have multiple bodies, each of which implements the same functionality with a different tradeoff of time versus space efficiency. Or, for machine-dependent code, there may be one body for each target machine. Maintaining multiple package specifications can also be useful during development and test. You may develop one specification for delivery to your customer and another for unit testing. The first one would export only those subprograms intended to be called from outside of the package during normal operation of the system. The second one would export all subprograms of the package so that each of them could be independently tested.

A consistent file naming convention is recommended to make it easier to manage the large number of files which may result from following this guideline.

4.1.2 Subprograms

guideline

- Use subprograms to enhance abstraction.
- Restrict each subprogram to the performance of a single action (NASA 1987).

example

Your program is required to output text to many types of devices. Because the devices would accept a variety of character sets, the proper way to do this is to write a subprogram to convert to the required character set within the subprogram that writes out the data. This way, the output subprogram has one purpose and the conversions are done elsewhere.

```
-------------------------------------------------------------------------------
procedure OUTPUT_TO_DEVICE (OUTPUT_DATA : in     TEXT_DATA;
                            DEVICE      : in     DEVICE_NAME;
                            STATUS      :    out ERROR_CODES) is
   -- local declarations
begin -- OUTPUT_TO_DEVICE
   ...
   case DEVICE.CHARACTER_SET is
      when LIMITED_ASCII  =>
         CONVERT_TO_UPPER_CASE (ORIGINAL_DATA   => OUTPUT_DATA,
                                UPPER_CASE_DATA => UPPER_OUTPUT_DATA);
         ...
      when EXTENDED_ASCII =>
         ...
      when EBCDIC         =>
         ...
   end case; -- DEVICE_TYPE.CHARACTER_SET
   ...
end OUTPUT_TO_DEVICE;
-------------------------------------------------------------------------------
```

rationale

Subprograms are an extremely effective and well-understood abstraction technique. Subprograms increase program readability by hiding the details of a particular activity. It is not necessary that a subprogram be called more than once to justify its existence.

The `case` statement in the example is more readable and understandable with the bodies of the conversion routines elsewhere, and the meaningful subprogram names enhance the understanding of the purpose of the `case` statement (see Guideline 3.2.4).

4.1.3 Functions

guideline

- When writing a function, make sure it has no side effects.

rationale

A side effect is a change to any variable that is not local to the subprogram. This includes changes to variables by other subprograms and entries during calls from the function if the changes persist after the function returns. Side effects are discouraged because they are difficult to understand and maintain. Additionally, the Ada language does not define the order in which functions are evaluated when they occur in expressions or as actual parameters to subprograms. Therefore, a program which depends on the order in which side effects of functions occur is erroneous. Avoid using side effects anywhere.

exception

There are a few cases in which functions with side effects are an accepted practice. One such case is a random number generator. Others, such as recording performance analysis data or information for recovery, have little to do with the application.

4.1.4 Packages

guideline

- Use packages for information hiding.
- Use packages with private types for abstract data types.
- Use packages to model abstract entities appropriate to the problem domain.
- Use packages to group together related type and object declarations (e.g., common declarations for two or more library units).
- Use packages to group together related program units for configuration control or visibility reasons (NASA 1987).
- Encapsulate machine dependencies in packages. Place a software interface to a particular device in a package to facilitate a change to a different device.

- Place low-level implementation decisions or interfaces in subprograms within packages.
- Use packages and subprograms to encapsulate and hide program details that may change (Nissen and Wallis 1984).

example

A package called BACKING_STORAGE_INTERFACE could contain type and subprogram declarations to support a generalized view of an external memory system (such as a disk or drum). Its internals may, in turn, depend on other packages more specific to the hardware or operating system.

rationale

Packages are the principal structuring facility in Ada. They are intended to be used as direct support for abstraction, information hiding, and modularization. For example, they are useful for encapsulating machine dependencies as an aid to portability. A single specification can have multiple bodies isolating implementation-specific information so other parts of the code do not need to change.

Encapsulating areas of potential change helps to minimize the effort required to implement that change by preventing unnecessary dependencies among unrelated parts of the system.

4.1.5 Functional Cohesion

guideline

- Make each package serve a single purpose.
- Use packages to group functionally related data, types, and subprograms.
- Avoid collections of unrelated objects and subprograms (NASA 1987 and Nissen and Wallis 1984).

example

As a bad example, a package named PROJECT_DEFINITIONS is obviously a "catch all" for a particular project and is likely to be a jumbled mess. It probably has this form to permit project members to incorporate a single `with` clause into their software.

Better examples are packages called DISPLAY_FORMAT_DEFINITIONS, containing all the types and constants needed by some specific display in a specific format, and CARTRIDGE_TAPE_HANDLER, containing all the types, constants, and subprograms which provide an interface to a special purpose device.

rationale

See also Guideline 5.4.1 on Heterogeneous Data.

The degree to which the entities in a package are related has a direct impact on the ease of understanding packages and programs made up of packages. There are different criteria for grouping, and some criteria are less effective than others. Grouping the class of data or activity (e.g., initialization modules) or grouping data or activities based on their timing characteristics is less effective than grouping based on function or need to communicate through data (Charrette 1986 paraphrased).

note

Traditional subroutine libraries often group functionally unrelated subroutines. Even such libraries should be broken into a collection of packages each containing a logically cohesive set of subprograms.

4.1.6 Data Coupling

guideline

- Avoid putting variables in package specifications.

example

This is part of a compiler. Both the package handling error messages and the package containing the code generator need to know the current line number. Rather than storing this in a shared variable of type NATURAL, the information is stored in a package that hides the details of how such information is represented, and makes it available with access routines.

```
-------------------------------------------------------------------------
package COMPILATION_STATUS is
   function SOURCE_LINE_NUMBER return LINE_RANGE;
end COMPILATION_STATUS;
-------------------------------------------------------------------------
with COMPILATION_STATUS;
package ERROR_MESSAGE_PROCESSING is
   -- Handle compile-time diagnostic.
end ERROR_MESSAGE_PROCESSING;
-------------------------------------------------------------------------
with COMPILATION_STATUS;
package CODE_GENERATION is
   -- Operations for code generation.
end CODE_GENERATION;
-------------------------------------------------------------------------
```

rationale

Strongly coupled program units can be difficult to debug and very difficult to maintain. By protecting shared data with access functions, the coupling is lessened. This prevents dependence on the data structure and access to the data can be controlled.

4.1.7 Tasks

guideline

- Use tasks to model abstract, asynchronous entities within the problem domain.
- Use tasks to control or synchronize access to tasks or other asynchronous entities (e.g., asynchronous I/O, peripheral devices, interrupts).
- Use tasks to define concurrent algorithms for multiprocessor architectures.
- Use tasks to perform concurrent, cyclic, or prioritized activities (NASA 1987).

rationale

The rationale for this guideline is given under Guideline 6.1.1. Chapter 6 discusses tasking in more detail.

4.2 VISIBILITY

Ada's ability to enforce information hiding and separation of concerns through its visibility controlling features is one of the most important advantages of the language, particularly when "pieces of a large system are being developed separately." Subverting these features, for example by excessive reliance on the use clause, is wasteful and dangerous. See also Section 5.7.

4.2.1 Minimization of Interfaces

guideline

- Put only what is needed for the use of a package into its specification.
- Minimize the declaration of objects in package specifications (Nissen and Wallis 1984).
- Do not include extra operations simply because they are easy to build.
- Minimize the context (`with`) clauses in a package specification.
- Reconsider subprograms which seem to require large numbers of parameters.
- Do not manipulate global data within a subprogram or package merely to limit the number of parameters.
- Avoid unnecessary visibility; hide the implementation details of a program unit from its users.

example

```
-------------------------------------------------------------------------------
package TELEPHONE_BOOK is

   type ENTRIES is limited private;
   procedure SET_NAME (...);

   procedure INSERT_ENTRY (...);
   procedure DELETE_ENTRY (...);
   -----------------------------------------------------------------------------
private
   type ENTRY_INFO;
   type ENTRIES is access ENTRY_INFO;
end TELEPHONE_BOOK;
-------------------------------------------------------------------------------
package body TELEPHONE_BOOK is

   type ENTRY_INFO is
      record
         ...     -- Full details of record for an entry
      end record;

   FIRST_ENTRY : ENTRIES;   ...
   -----------------------------------------------------------------------------
   procedure INSERT_ENTRY (...) is
   begin
      ...
   end INSERT_ENTRY;
   -----------------------------------------------------------------------------
   procedure DELETE_ENTRY (...) is
   begin
      ...
   end DELETE_ENTRY;
   -----------------------------------------------------------------------------
end TELEPHONE_BOOK;
-------------------------------------------------------------------------------
```

rationale

For each entity in the specification, give careful consideration to whether it could be moved to the body. The fewer the extraneous details, the more understandable the program, package, or subprogram. It is important to maintainers to know exactly what a package interface is so that they can understand the effects of changes. Interfaces to a subprogram extend beyond the parameters. Any modification of global data from within a package or subprogram is an undocumented interface to the "outside" as well.

Pushing as many as possible of the context dependencies into the body makes the reader's job easier, localizes the recompilation required when library units change, and helps prevent a ripple effect during modifications. See also Guideline 4.2.3.

Subprograms with large numbers of parameters often indicate poor design decisions (e.g., the functional boundaries of the subprogram are inappropriate, or parameters are structured poorly). Conversely, subprograms with no parameters are likely to be accessing global data.

Objects visible within package specifications can be modified by any unit that has visibility to them. The object cannot be protected or represented abstractly by its enclosing package. Objects which must persist should be declared in package bodies. Objects whose value depends on program units external to their enclosing package are probably either in the wrong package or are better accessed by a subprogram specified in the package specification.

note

The specifications of some packages, such as Ada bindings to existing subroutine libraries, cannot easily be reduced in size. In such cases, it may be beneficial to break these up into smaller packages, grouping according to category (e.g., trigonometric functions).

4.2.2 Nested Packages

guideline

- Nest package specifications within another package specification only for grouping operations, hiding common implementation details, or presenting different views of the same abstraction.

example

Chapter 14 of the Ada LRM (Ada Reference Manual 1983) gives an example of desirable package specification nesting. The specifications of generic packages INTEGER_IO, FLOAT_IO, FIXED_IO, and ENUMERATION_IO are nested within the specification of package TEXT_IO. Each of them is a generic, grouping closely related operations and needing to use hidden details of the implementation of TEXT_IO.

rationale

Grouping package specifications into an encompassing package emphasizes a relationship of commonality among those packages. It also allows them to share common implementation details resulting from the relationship.

An abstraction occasionally needs to present different views to different classes of users. Building one view upon another as an additional abstraction does not always suffice, because the functionality of the operations presented by the views may be only partially disjoint. Nesting specifications groups the facilities of the various views, yet associates them with the abstraction they present. Abusive mixing of the views by another unit would be easy to detect due to the multiple use clauses or an incongruous mix of qualified names.

4.2.3 Restricting Visibility

guideline

- Restrict the visibility of program units as much as possible by nesting them inside other program units and hiding them inside package bodies (Nissen and Wallis 1984).
- Minimize the scope within which `with` clauses apply.
- Only `with` those units directly needed.

example

This program is a compiler. Access to the printing facilities of TEXT_IO is restricted to the software involved in producing the source code listing.

```
-------------------------------------------------------------------------
procedure COMPILER is
   ----------------------------------------------------------------------
   package LISTING_FACILITIES is

      procedure NEW_PAGE_OF_LISTING;
      procedure NEW_LINE_OF_PRINT;
      -- etc.

   end LISTING_FACILITIES;
   ----------------------------------------------------------------------
   package body LISTING_FACILITIES  is separate;
   ----------------------------------------------------------------------
begin -- COMPILER
   ...
end COMPILER;
-------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------
with TEXT_IO;
   separate (COMPILER)
package body LISTING_FACILITIES is
   -------------------------------------------------------------------------
   procedure NEW_PAGE_OF_LISTING is
   begin
      ...
   end NEW_PAGE_OF_LISTING;
   -------------------------------------------------------------------------
   procedure NEW_LINE_OF_PRINT is
   begin
      ...
   end NEW_LINE_OF_PRINT;
   -------------------------------------------------------------------------
   -- etc

end LISTING_FACILITIES;
-------------------------------------------------------------------------
```

rationale

Restricting visibility of a program unit ensures that the program unit is not called from some other part of the system than that which was intended. This is done by nesting it inside of the only unit which uses it, or by hiding it inside of a package body rather than declaring it in the package specification. This avoids errors and eases the job of maintainers by guaranteeing that a local change in that unit will not have an unforeseen global effect.

Restricting visibility of a library unit, by using `with` clauses on subunits rather than on the entire parent unit, is useful in the same way. In the example above, it is clear that the package `TEXT_IO` is used only by the `LISTING_FACILITIES` package of the compiler.

note

One way to minimize the coverage of a `with` clause is to use it only with subunits that really need it. Consider making them subunits when the need for visibility to a library unit is restricted to a subprogram or two.

4.2.4 Hiding Tasks

guideline

- Carefully consider encapsulation of tasks.

example

```
-------------------------------------------------------------------------
package DISK_HEAD_SCHEDULER is
   type TRACK_NUMBER is ...
   type WORDS is ...
   -------------------------------------------------------------------------
   procedure TRANSMIT (TRACK : in TRACK_NUMBER;
                       DATA  : in WORDS);
   -------------------------------------------------------------------------
   ...
end DISK_HEAD_SCHEDULER;
-------------------------------------------------------------------------
package body DISK_HEAD_SCHEDULER is
   ...
   -------------------------------------------------------------------------
   task CONTROL is
      entry SIGN_IN (TRACK : in TRACK_NUMBER);
      ...
   end CONTROL;
   -------------------------------------------------------------------------
   task TRACK_MANAGER is
      entry TRANSFER (TRACK_NUMBER) (DATA : in WORDS);
   end TRACK_MANAGER;
   -------------------------------------------------------------------------
   ...
   -------------------------------------------------------------------------
   procedure TRANSMIT (TRACK : in TRACK_NUMBER;
                       DATA  : in WORDS) is
   begin
      CONTROL.SIGN_IN (TRACK);
      TRACK_MANAGER.TRANSFER (TRACK) (DATA);
   end TRANSMIT;
   -------------------------------------------------------------------------
   ...
end DISK_HEAD_SCHEDULER;
-------------------------------------------------------------------------
```

rationale

The decision whether to declare a task in the specification or body of an enclosing package is not a simple one. There are good arguments for both.

Hiding a task specification in a package body and exporting (via subprograms) only required entries reduces the amount of extraneous information in the package specification. It allows your subprograms to enforce any order of entry calls necessary to the proper operation of the tasks. It also allows you to impose defensive task communication practices (see Guideline 6.2.2) and proper use of conditional and timed entry calls. Finally, it allows the grouping of entries into sets for export to different classes of users (e.g., producers versus consumers), or the concealment of entries that should not be made public at all (e.g., initialization, completion, signals). Where performance is an issue and there are no ordering rules to enforce, the entries can be renamed as subprograms to avoid the overhead of an extra procedure call.

An argument which can be viewed as an advantage or disadvantage is that hiding the task specification in a package body hides the fact of a tasking implementation from the user. If the application is such that a change to or from a tasking implementation, or a reorganization of services among tasks, need not concern users of the package then this is an advantage. However, if the package user must know about the tasking implementation to reason about global tasking behavior, then it is better not to hide the task completely. Either move it to the package specification or add comments stating that there is a tasking implementation, describing when a call may block, etc. Otherwise, it is the package implementor's responsibility to ensure that users of the package do not have to concern themselves with behaviors such as deadlock, starvation, and race conditions.

Finally, keep in mind that hiding tasks behind a procedural interface prevents the usage of conditional and timed entry calls and entry families, unless you add parameters and extra code to the procedures to make it possible for callers to direct the procedures to use these capabilities.

4.3 EXCEPTIONS

This section addresses the issue of exceptions in the context of program structures. It discusses how exceptions should be used as part of the interface to a unit, including what exceptions to declare and raise and under what conditions to raise them. Information on how to handle, propagate, and avoid raising exceptions is found in Section 5.8.

4.3.1 Using Exceptions as Help Define an Abstraction

guideline

- Declare a different exception name for each error that the user of a unit can make.
- Declare a different exception name for each unavoidable and unrecoverable internal error which can occur in a unit.
- Do not borrow an exception name from another context.
- Export (declare visibly to the user) the names of all exceptions which can be raised.
- In a package, document which exceptions can be raised by each subprogram and task entry.
- Do not raise exceptions for internal errors which can be avoided or corrected within the unit.
- Do not raise the same exception to report different types of errors which are distinguishable by the user of the unit.
- Provide interrogative functions which allow the user of a unit to avoid causing exceptions to be raised.
- When possible, avoid changing state information in a unit before raising an exception.
- Catch and convert or handle all predefined and compiler-defined exceptions at the earliest opportunity.
- Do not explicitly raise predefined or implementation-defined exceptions.
- Never let an exception propagate beyond its scope.

example

This package specification defines an exception which enhances the abstraction:

```
-------------------------------------------------------------------------
package STACK is

   function STACK_EMPTY return BOOLEAN;

   NO_DATA_ON_STACK : exception;
      -- Raised when POP is used on empty stack.
   procedure POP (...);
   procedure PUSH (...);

end STACK;
-------------------------------------------------------------------------
```

This example shows how to convert a predefined exception to a user-defined one:

```
   procedure POP (...) is                RETURN_VALUE  := STACK_POINTER.DATA;
STACK_POINTER := STACK_POINTER.NEXT;       return RETURN_VALUE;   exception      when
CONSTRAINT_ERROR =>          raise NO_DATA_ON_STACK;   end POP;
```

rationale

Exceptions should be used as part of an abstraction to indicate error conditions which the abstraction is unable to prevent or correct. Since the abstraction is unable to correct such an error, it must report the error to the user. In the case of a usage error (e.g., attempting to invoke operations in the wrong sequence or attempting to exceed a boundary condition), the user may be able to correct the error. In the case of an error beyond the control of the user, the user may be able to work around the error if there are multiple mechanisms available to perform the desired operation. In other cases, the user may have to abandon use of the unit, dropping into a degraded mode of limited functionality. In any case, the user must be notified.

Exceptions are a good mechanism for reporting such errors because they provide an alternate flow of control for dealing with errors. This allows error-handling code to be kept separate from the code for normal processing. When an exception is raised, the current operation is aborted and control is transferred directly to the appropriate exception handler.

Several of the guidelines above exist to maximize the ability of the user to distinguish and correct different types of errors. Providing a different exception name for each error condition makes it possible to handle each error condition separately. Declaring new exception names, rather than raising exceptions declared in other packages, reduces the coupling between packages and also makes different exceptions more distinguishable. Exporting the names of all exceptions which a unit can raise, rather than declaring them internally to the unit, makes it possible for users of the unit to refer to the names in exception handlers. Otherwise, the user would be able to handle the exception only with an `others` handler. Finally, using comments to document exactly which of the exceptions declared in a package can be raised by each subprogram or task entry making it possible for the user to know which exception handlers are appropriate in each situation.

Because they cause an immediate transfer of control, exceptions are useful for reporting unrecoverable errors which prevent an operation from being completed, but not for reporting status or modes incidental to the completion of an operation. They should not be used to report internal errors which a unit was able to correct invisibly to the user.

To provide the user with maximum flexibility, it is a good idea to provide interrogative functions which the user can call to determine whether an exception would be raised if a subprogram or task entry were invoked. The function STACK_IS_EMPTY in the above example is such a function. It indicates whether NO_DATA_ON_STACK would be raised if POP were called. Providing such functions makes it possible for the user to avoid triggering exceptions.

To support error recovery by its user, a unit should try to avoid changing state during an invocation which raises an exception. If a requested operation cannot be completely and correctly performed, then the unit should either detect this before changing any internal state information, or should revert back to the state at the time of the request. For example, after raising the exception NO_DATA_ON_STACK, the stack package in the above example should remain in exactly the same state it was in when POP was called. If it were to partially update its internal data structures for managing the stack, then future PUSH and POP operations would not perform correctly. This is always desirable, but not always possible.

User-defined exceptions should be used instead of predefined or compiler-defined exceptions because they are more descriptive and more specific to the abstraction. The predefined exceptions are very general, and can be triggered by many different situations. Compiler-defined exceptions are nonportable and have meanings which are subject to change even between successive releases of the same compiler. This introduces too much uncertainty for the creation of useful handlers.

If you are writing an abstraction, remember that the user does not know about the units you use in your implementation. That is an effect of information hiding. If any exception is raised within your abstraction, you must catch it and handle it. The user is not able to provide a reasonable handler if the original exception is allowed to propagate out. You can still convert the exception into a form intelligible to the user if your abstraction cannot effectively recover on its own.

Converting an exception means raising a user-defined exception in the handler for the original exception. This introduces a meaningful name for export to the user of the unit. Once the error situation is couched in terms of the application, it can be handled in those terms.

Do not allow an exception to propagate unhandled outside the scope of the declaration of its name, because then only a handler for `others` can catch it. As discussed under Guideline 5.8.2, a handler for `others` cannot be written to deal with the specific error effectively.

4.4 SUMMARY

high-level structure

- Place the specification of each library unit package in a separate file from its body.
- Create an explicit specification, in a separate file, for each library unit subprogram.
- Use subunits for the bodies of large units which are nested in other units.
- Place each subunit in a separate file.

- Use a consistent file naming convention.
- Use subprograms to enhance abstraction.
- Restrict each subprogram to the performance of a single action.
- When writing a function, make sure it has no side effects.
- Use packages for information hiding.
- Use packages with private types for abstract data types.
- Use packages to model abstract entities appropriate to the problem domain.
- Use packages to group together related type and object declarations (e.g., common declarations for two or more library units).
- Use packages to group together related program units for configuration control or visibility reasons.
- Encapsulate machine dependencies in packages. Place a software interface to a particular device in a package to facilitate a change to a different device.
- Place low-level implementation decisions or interfaces in subprograms within packages.
- Use packages and subprograms to encapsulate and hide program details that may change.
- Make each package serve a single purpose.
- Use packages to group functionally related data, types, and subprograms.
- Avoid collections of unrelated objects and subprograms.
- Avoid putting variables in package specifications.
- Use tasks to model abstract, asynchronous entities within the problem domain.
- Use tasks to control or synchronize access to tasks or other asynchronous entities (e.g., asynchronous I/O, peripheral devices, interrupts).
- Use tasks to define concurrent algorithms for multiprocessor architectures.
- Use tasks to perform concurrent, cyclic, or prioritized activities.

visibility

- Put only what is needed for the use of a package into its specification.
- Minimize the declaration of objects in package specifications.
- Do not include extra operations simply because they are easy to build.
- Minimize the context (`with`) clauses in a package specification.
- Reconsider subprograms which seem to require large numbers of parameters.
- Do not manipulate global data within a subprogram or package merely to limit the number of parameters.
- Avoid unnecessary visibility; hide the implementation details of a program unit from its users.
- Nest package specifications within another package specification only for grouping operations, hiding common implementation details, or presenting different views of the same abstraction.
- Restrict the visibility of program units as much as possible by nesting them inside other program units and hiding them inside package bodies.
- Minimize the scope within which `with` clauses apply.
- Only `with` those units directly needed.
- Carefully consider encapsulation of tasks.

exceptions

- Declare a different exception name for each error that the user of a unit can make.
- Declare a different exception name for each unavoidable and unrecoverable internal error which can occur in a unit.

- Do not borrow an exception name from another context.
- Export (declare visibly to the user) the names of all exceptions which can be raised.
- In a package, document which exceptions can be raised by each subprogram and task entry.
- Do not raise exceptions for internal errors which can be avoided or corrected within the unit.
- Do not raise the same exception to report different types of errors which are distinguishable by the user of the unit.
- Provide interrogative functions which allow the user of a unit to avoid causing exceptions to be raised.
- When possible, avoid changing state information in a unit before raising an exception.
- Catch and convert or handle all predefined and compiler-defined exceptions at the earliest opportunity.
- Do not explicitly raise predefined or implementation-defined exceptions.
- Never let an exception propagate beyond its scope.

CHAPTER 5
Programming Practices

Software is always subject to change. The need for this change, euphemistically known as "maintenance" arises from a variety of sources. Errors need to be corrected as they are discovered. System functionality may need to be enhanced in planned or unplanned ways. Inevitably, the requirements change over the lifetime of the system, forcing continual system evolution. Often, these modifications are conducted long after the software was originally written, usually by someone other than the original author.

Easy and successful modification requires that the software be readable, understandable, and structured according to accepted practice. If a software component cannot be understood easily by a programmer who is familiar with its intended function, that *software component is not maintainable*. Techniques that make code readable and comprehensible enhance its maintainability. So far, we have visited such techniques as consistent use of naming conventions, clear and well-organized commentary, and proper modularization. We now present consistent and logical use of language features.

Correctness is one aspect of reliability. While style guidelines cannot enforce the use of correct algorithms, they can suggest the use of techniques and language features known to reduce the number or likelihood of failures. Such techniques include program construction methods that reduce the likelihood of errors or that improve program predictability by defining behavior in the presence of errors.

5.1 OPTIONAL PARTS OF THE SYNTAX

Parts of the Ada syntax, while optional, can enhance the readability of the code. The guidelines given below concern use of some of these optional features.

5.1.1 Loop Names

guideline

- Associate names with loops when they are nested (Booch 1987, 195).

example

```
DOCUMENT_PAGES:
   loop
      ...
      PAGE_LINES:
         loop
            ...
            exit PAGE_LINES when LINE_NUMBER = MAX_LINES_ON_PAGE;
            ...
            LINE_SYMBOLS:
               loop
                  ...
                  exit LINE_SYMBOLS when CURRENT_SYMBOL = SENTINEL;
                  ...
               end loop LINE_SYMBOLS;
            ...
         end loop PAGE_LINES;
      ...
      exit DOCUMENT_PAGES when PAGE_NUMBER = MAXIMUM_PAGES;
      ...
   end loop DOCUMENT_PAGES;
```

rationale

When you associate a name with a loop, you must include that name with the associated end for that loop (Ada Reference Manual 1983). This helps readers find the associated end for any given loop. This is especially true if loops are broken over screen or page boundaries. The choice of a good name for the loop documents its purpose, reducing the need for explanatory comments. If a name for a loop is very difficult to choose, this could indicate a need for more thought about the algorithm.

Regularly naming loops helps you follow Guideline 5.1.3.

It can be difficult to think up a name for every loop, therefore the guideline specifies nested loops. The benefits in readability and second thought outweigh the inconvenience of naming the loops.

5.1.2 Block Names

guideline

- Associate names with blocks when they are nested.

example

```
TRIP:
   declare
      -- local object declarations
   begin
      ARRIVE_AT_AIRPORT:
         declare
            -- local object declarations
         begin
            -- Activities to RENT_CAR,
            -- Activities to CLAIM_BAGGAGE,
            -- Activities to RESERVE_HOTEL.
            -- Exception handlers, etc.
         end ARRIVE_AT_AIRPORT;

      VISIT_CUSTOMER:
         declare
            -- local object declarations
         begin
            -- again a set of activities...
            -- exception handlers, etc.
         end VISIT_CUSTOMER;

      DEPARTURE_PREPARATION:
         declare
            -- local object declarations
         begin
            -- Activities to RETURN_CAR,
            -- Activities to CHECK_BAGGAGE,
            -- Activities to WAIT_FOR_FLIGHT.
            -- Exception handlers, etc.
         end DEPARTURE_PREPARATION;

      BOARD_RETURN_FLIGHT;
   end TRIP;
```

rationale

When there is a nested block structure it can be difficult to determine which `end` corresponds to which block. Naming blocks alleviates this confusion. The choice of a good name for the block documents its purpose, reducing the need for explanatory comments. If a name for the block is very difficult to choose, this could indicate a need for more thought about the algorithm.

This guideline is also useful if nested blocks are broken over a screen or page boundary.

It can be difficult to think up a name for each block, therefore the guideline specifies nested blocks. The benefits in readability and second thought outweigh the inconvenience of naming the blocks.

5.1.3 Exit Statements

guideline

- Use loop names on all exit statements.

example

See the example in Section 5.1.1.

rationale

An exit statement is an implicit `goto`. It should specify its source explicitly. When there is a nested loop structure and an exit statement is used, it can be difficult to determine which loop is being exited. Also, future changes which may introduce a nested loop are likely to introduce an error, with the exit accidentally exiting from the wrong loop. Naming loops and their exits alleviates this confusion. This guideline is also useful if nested loops are broken over a screen or page boundary.

5.1.4 Naming End Statements

guideline

- Include the simple name at the end of a package specification and body.

- Include the simple name at the end of a task specification and body.
- Include the simple name at the end of an accept statement.
- Include the designator at the end of a subprogram body.

example

```
-------------------------------------------------------------------------
package AUTOPILOT is
   ...
   function IS_ENGAGED ... ;
   ...
   procedure DISENGAGE ... ;
   ...
end AUTOPILOT;
-------------------------------------------------------------------------
package body AUTOPILOT is
   ----------------------------------------------------------------------
   task type COURSE_MONITOR is
      ...
      entry RESET ... ;
      ...
   end COURSE_MONITOR;
   ----------------------------------------------------------------------
   function IS_ENGAGED ... is
      ...
   end IS_ENGAGED;
   ----------------------------------------------------------------------
   procedure DISENGAGE ... is
      ...
   end DISENGAGE;
   ----------------------------------------------------------------------
   task body COURSE_MONITOR is
      ...
      accept RESET ... do
         ...
      end RESET;
      ...
   end COURSE_MONITOR;
   ----------------------------------------------------------------------
end AUTOPILOT;
-------------------------------------------------------------------------
```

rationale

Repeating names on the end of these compound statements ensures consistency throughout the code. In addition, the named end provides a reference for the reader if the unit spans a page or screen boundary, or if it contains a nested unit.

5.2 PARAMETER LISTS

A subprogram or entry parameter list is the interface to the abstraction implemented by the subprogram or entry. It is important that it is clear, and is expressed in a consistent style. Careful decisions about formal parameter naming and ordering can make the purpose of the subprogram easier to understand which can make it easier to use.

5.2.1 Formal Parameters

guideline

- Name formal parameters descriptively to reduce the need for comments.

example

```
LIST_MANAGER.INSERT
   (ELEMENT     => EMPLOYEE_RECORD,
    INTO_LIST   => LIST_OF_EMPLOYEES,
    AT_POSITION => 1);
```

rationale

Following the variable naming guidelines (Guidelines 3.2.1 and 3.2.3) for formal parameters can make calls to subprograms read more like regular prose, as shown in the example above where no comments are necessary. Descriptive names of this sort can also make the code in the body of the subprogram more clear.

5.2.2 Named Association

guideline

- Use named parameter association in calls of infrequently used subprograms or entries with many formal parameters.
- Use named association for constants, expressions, and literals in aggregates.
- Use named association when instantiating generics.
- Use named association for clarification when the actual parameter is any literal or expression.
- Use named association when supplying a nondefault value to an optional parameter.

instantiation

- Use named parameter association in calls of subprograms or entries called from less than five places in a single source file or with more than two formal parameters.

example

```
ENCODE_TELEMETRY_PACKET
  (SOURCE         => POWER_ELECTRONICS;
   CONTENT        => TEMPERATURE;
   VALUE          => READ_TEMPERATURE_SENSOR (POWER_ELECTRONICS);
   TIME           => CURRENT_TIME;
   SEQUENCE       => NEXT_PACKET_ID;
   VEHICLE        => THIS_SPACECRAFT;
   PRIMARY_MODULE => TRUE);
```

rationale

Calls of infrequently used subprograms or entries with many formal parameters can be difficult to understand without referring to the subprogram or entry code. Named parameter association can make these calls more readable.

When the formal parameters have been named appropriately, it is easier to determine exactly what purpose the subprogram serves without looking at its code. This reduces the need for named constants that exist solely to make calls more readable. It also allows variables used as actual parameters to be given names indicating what they are without regard to why they are being passed in a call. An actual parameter, which is an expression rather than a variable, cannot be named otherwise.

Named association allows subprograms to have new parameters inserted with minimal ramifications to existing calls.

note

The judgment of when named parameter association improves readability is subjective. Certainly, simple or familiar subprograms such as a swap routine or a sine function do not require the extra clarification of named association in the procedure call.

caution

A consequence of named parameter association is that the formal parameter names may not be changed without modifying the text of each call.

5.2.3 Default Parameters

guideline

- Provide default parameters to allow for occasional, special use of widely used subprograms or entries.
- Place default parameters at the end of the formal parameter list.

- Consider providing default values to new parameters added to an existing subprogram.

example

Chapter 14 of the Ada Language Reference Manual (Department of Defense 1983) contains many examples of this practice.

rationale

Often, the majority of uses of a subprogram or entry need the same value for a given parameter. Providing that value, as the default for the parameter, makes the parameter optional on the majority of calls. It also allows the remaining calls to customize the subprogram or entry by providing different values for that parameter.

Placing default parameters at the end of the formal parameter list allows the caller to use positional association on the call, otherwise defaults are available only when named association is used.

Often during maintenance activities, you increase the functionality of a subprogram or entry. This requires more parameters than the original form for some calls. New parameters may be required to control this new functionality. Give the new parameters default values which specify the old functionality. Calls needing the old functionality need not be changed; they take the defaults. This is true if the new parameters are added to the end of the parameter list, or if named association is used on all calls. New calls needing the new functionality can specify that by providing other values for the new parameters.

This enhances maintainability in that the places which use the modified routines do not themselves have to be modified, while the previous functionality levels of the routines are allowed to be "reused."

exceptions

Do not go overboard. If the changes in functionality are truly radical, you should be preparing a separate routine rather than modifying an existing one. One indicator of this situation would be that it is difficult to determine value combinations for the defaults that uniquely and naturally require the more restrictive of the two functions. In such cases, it is better to go ahead with creation of a separate routine.

5.2.4 Mode Indication

guideline

- Show the mode indication of all procedure and entry parameters (Nissen and Wallis 1984).
- Select the most restrictive mode possible.

example

```
procedure OPEN_FILE (FILE_NAME   : in     SPC_STRING;
                     OPEN_STATUS :    out STATUS_CODES);

entry     ACQUIRE   (KEY         : in     CAPABILITY;
                     RESOURCE    :    out TAPE_DRIVE);
```

rationale

By showing the mode of parameters, you aid the reader. If you do not specify a parameter mode, the default mode is `in`. Explicitly showing the mode indication of all parameters is a more assertive action than simply taking the default mode. Anyone reviewing the code later will be more confident that you intended the parameter mode to be `in`.

Use the mode that reflects the actual use of the parameter. Only use `in out` mode when reading and writing to a parameter.

5.2.5 Order of Parameter Declarations

guideline

- Declare parameters in a consistent order (Honeywell 1986).

instantiation

- All `in` parameters without default values are declared before any `in out` parameter.

- All `in out` parameters are declared before any `out` parameters.
- All `out` parameters are declared before any parameters with default values.
- All parameters with default values are declared last.
- The order of parameters within these groups is derived from the needs of the application.

example

```
procedure ASSEMBLE_TELEMETRY_MESSAGE
   (INPUT_PACKET                 : in     TELEMETRY_PACKET;
    TRANSFER_TO_DOWNLINK_BUFFER  : in out PACKET_BUFFER;
    BUFFER_FULL                  :    out BOOLEAN);
```

rationale

By declaring all the parameters in a consistent order, you make the code easier to read and understand. Some of your choices are to arrange the parameters by `mode` or to group the parameters by function. Also, grouping default parameters at the end of the list allows calls to be made using positional notation and the default values.

In special cases, parameters declared in a nonstandard order may be more readable. Since consistency is the goal, however, the readability or some other quality of the code must be enhanced if you deviate from this guideline. Default parameters are an example of the need for a special case.

5.3 TYPES

In addition to determining the possible values for variables, type names, and distinctions can be very valuable aids in developing safe, readable, and understandable code. Types clarify the structure of your data and can limit or restrict the operations that can be performed on that data. "Keeping types distinct has been found to be a very powerful means of detecting logical mistakes when a program is written and to give valuable assistance whenever the program is being subsequently maintained" (Pyle 1985). Take advantage of Ada's strong typing capability in the form of subtypes, derived types, task types, private types, and limited private types.

The guidelines encourage much code to be written to ensure strong typing (i.e., subtypes). While it might appear that there would be execution penalties for this amount of code, this is usually not the case. In contrast to other conventional languages, Ada has a less direct relationship between the amount of code that is written and the size of the resulting executable program. Most of the strong type checking is performed at compilation time rather than execution time, so the size of the executable code is not greatly affected.

5.3.1 Derived Types and Subtypes

guideline

- Use existing types as building blocks by deriving new types from them.
- Use range constraints on subtypes.
- Define new types, especially derived types, to include the largest set of possible values, including boundary values.
- Constrain the ranges of derived types with subtypes, excluding boundary values.

example

Type `TABLE` is a building block for the creation of new types:

```
type TABLE is
  record
     COUNT : LIST_SIZE  := EMPTY;
     LIST  : ENTRY_LIST := EMPTY_LIST;
  end record;

type TELEPHONE_DIRECTORY  is new TABLE;
type DEPARTMENT_INVENTORY is new TABLE;
```

The following are distinct types that cannot be intermixed in operations that are not programmed explicitly to use them both:

```
type DOLLARS is new NUMBER;
type CENTS   is new NUMBER;
```

Below, SOURCE_TAIL has a value outside the range of LISTING_PAPER when the line is empty. All the indices can be mixed in expressions, as long as the results fall within the correct subtypes:

```
type    COLUMNS         is range FIRST_COLUMN - 1 .. LISTING_WIDTH + 1;
subtype LISTING_PAPER   is COLUMNS
                           range FIRST_COLUMN      .. LISTING_WIDTH;
subtype DUMB_TERMINAL   is COLUMNS
                           range FIRST_COLUMN      .. DUMB_TERMINAL_WIDTH;

type    LISTING_LINE    is array (LISTING_PAPER) of BYTES;
type    TERMINAL_LINE   is array (DUMB_TERMINAL) of BYTES;

SOURCE_TAIL             : COLUMNS         := COLUMNS'FIRST;
SOURCE                  : LISTING_LINE;
DESTINATION             : TERMINAL_LINE;
...
DESTINATION (DESTINATION'FIRST .. (SOURCE_TAIL - DESTINATION'LAST)) :=
     SOURCE (COLUMNS'SUCC (DESTINATION'LAST) .. SOURCE_TAIL);
```

rationale

The name of a derived type can make clear its intended use and avoid proliferation of similar type definitions. Objects of two derived types, even though derived from the same type, cannot be mixed in operations unless such operations are supplied explicitly or one is converted to the other explicitly. This prohibition is an enforcement of strong typing.

Define new types, derived types, and subtypes cautiously and deliberately. The concepts of subtype and derived type are not equivalent, but they can be used to advantage in concert. A subtype limits the range of possible values for a type, but does not define a new type.

Types can have highly constrained sets of values without eliminating useful values. Used in concert, derived types and subtypes can eliminate many flag variables and type conversions within executable statements. This renders the program more readable, enforces the abstraction, and allows the compiler to enforce strong typing constraints.

Many algorithms begin or end with values just outside the normal range. If boundary values are not compatible within subexpressions, algorithms can be needlessly complicated. The program can become cluttered with flag variables and special cases when it could just test for zero or some other sentinel value just outside normal range.

The derived type COLUMNS and the subtype LISTING_PAPER in the example above demonstrate how to allow sentinel values. The subtype LISTING_PAPER could be used as the type for parameters of subprograms declared in the specification of a package. This would restrict the range of values which could be specified by the caller. Meanwhile, the derived type COLUMNS could be used to store such values internally to the body of the package, allowing FIRST_COLUMN - 1 to be used as a sentinel value. This combination of derived types and subtypes allows compatibility between subtypes within subexpressions without type conversions as would happen with derived types.

note

The price of the reduction in the number of independent type declarations is that subtypes and derived types change when the base type is redefined. This trickle-down of changes is sometimes a blessing and sometimes a curse. However, usually it is intended and beneficial.

5.3.2 Anonymous Types

guideline

- Do not use anonymous types.

example

```
-- Use
type BUFFER is array (BUFFER_INDEX) of CHARACTER;
INPUT_LINE : BUFFER;
-- rather than
INPUT_LINE : array (BUFFER_INDEX) of CHARACTER;
```

rationale

Although Ada allows anonymous types, they have limited usefulness and complicate program modification. For example, a variable of anonymous type can never be used as an actual parameter because it is not possible to define a formal parameter of the same type. Even though this may not be a limitation initially, it precludes a modification in which a piece of code is changed to a subprogram. Also, two variables declared using the same anonymous type declaration are actually of different types.

note

For task types, see Guideline 6.1.2. For unconstrained arrays as formal parameters, see Guideline 8.2.2.

In reading the Ada Language Reference Manual (Department of Defense 1983), you will notice that there are cases when anonymous types are mentioned abstractly as part of the description of the Ada computational model. These cases do not violate this guideline.

5.3.3 Private Types

guideline

- Use limited private types in preference to private types.
- Use private types in preference to nonprivate types.
- Explicitly export needed operations rather than easing restrictions.

example

```
-------------------------------------------------------------------------
package PACKET_TELEMETRY is
   type FRAME_HEADER is limited private;
   type FRAME_DATA is private;
   type FRAME_CODES is
      (MAIN_BUS_VOLTAGE, TRANSMITTER_1_POWER,
      ... );
   ...
   -------------------------------------------------------------------------
private
   type FRAME_HEADER is
      record
         ...;
      end record;

   type FRAME_DATA is
      record
         ...
      end record;
   ...
end PACKET_TELEMETRY;
-------------------------------------------------------------------------
```

rationale

Limited private types and private types support abstraction and information hiding better than nonprivate types. The more restricted the type, the better information hiding is served. This, in turn, allows the implementation to change without affecting the rest of the program. While there are many valid reasons to export types, it is better to try the preferred route first, loosening the restrictions only as necessary. If it is necessary for a user of the package to use a few of the restricted operations, it is better to export the operations explicitly and individually via exported subprograms than to drop a level of restriction. This practice retains the restrictions on other operations.

Limited private types have the most restricted set of operations available to users of a package. Of the types that must be made available to users of a package, as many as possible should be limited private. The operations available to limited private types are membership tests, selected components, components for the selections of any discriminant, qualification and explicit conversion, and attributes 'BASE and 'SIZE. Objects of a limited private type also have the attribute 'CONSTRAINED if there are discriminants. None of these operations allow the user of the package to manipulate objects in a way that depends on the structure of the type.

If additional operations must be available to the type, the restrictions may be loosened by making it a private type. The operations available on objects of private types that are not available on objects of limited private types are assignment and tests for equality and inequality. There are advantages to the restrictive nature of limited private types. For example, assignment allows copies of an object to be made. This could be a problem if the object's type is a pointer.

5.4 DATA STRUCTURES

The data structuring capabilities of Ada are a powerful resource; therefore, use them to model the data as closely as possible. It is possible to group logically related data and let the language control the abstraction and operations on the data rather than requiring the programmer or maintainer to do so. Data can also be organized in a building block fashion. In addition to showing how a data structure is organized (and possibly giving the reader an indication as to why it was organized that way), creating the data structure from smaller components allows those components to be reused themselves. Using the features that Ada provides can increase the maintainability of your code.

5.4.1 Heterogeneous Data

guideline

- Use records to group heterogeneous but related data.
- Consider records to map to I/O device data.

example

```
type PROPULSION_METHOD is (SAIL, DIESEL, NUCLEAR);
type CRAFT is
   record
      NAME   : STRING ( ... );
      PLANT  : PROPULSION_METHOD;
      LENGTH : FEET;
      BEAM   : FEET;
      DRAFT  : FEET;
   end record;
type FLEET is array ( ... ) of CRAFT;
```

rationale

You help the maintainer find all of the related data by gathering it into the same construct, simplifying any modifications that apply to all rather than part. This in turn increases reliability. Neither you nor an unknown maintainer are liable to forget to deal with all the pieces of information in the executable statements, especially if updates are done with aggregate assignments whenever possible.

The idea is to put the information a maintainer needs to know where it can be found with the minimum of effort. For example, if all information relating to a given CRAFT is in the same place, the relationship is clear both in the declarations and especially in the code accessing and updating that information. But, if it is scattered among several data structures, it is less obvious that this is an intended relationship as opposed to a coincidental one. In the latter case, the declarations may be grouped together to imply intent, but it may not be possible to group the accessing and updating code that way. Ensuring the use of the same index to access the corresponding element in each of several parallel arrays is difficult if the accesses are at all scattered.

If the application must interface directly to hardware, the use of records, especially in conjunction with record representation clauses, could be useful to map onto the layout of the hardware in question.

note

It may seem desirable to store heterogeneous data in parallel arrays in what amounts to a FORTRAN-like style. This style is an artifact of FORTRAN's data structuring limitations. FORTRAN only has facilities for constructing homogeneous arrays. Ada's record types offer one way to specify what are called nonhomogeneous arrays or heterogeneous arrays.

exceptions

If the application must interface directly to hardware, and the hardware requires that information be distributed among various locations, then it may not be possible to use records.

5.4.2 Nested Records

guideline

- Record structures should not always be flat. Factor out common parts.
- For a large record structure, group related components into smaller subrecords.
- For nested records, pick element names that read well when inner elements are referenced.

example

```
type COORDINATE is
   record
      ROW    : LOCAL_FLOAT;
      COLUMN : LOCAL_FLOAT;
   end record;

type WINDOW is
   record
      TOP_LEFT     : COORDINATE;
      BOTTOM_RIGHT : COORDINATE;
   end record;
```

rationale

You can make complex data structures understandable and comprehensible by composing them of familiar building blocks. This technique works especially well for large record types with parts which fall into natural groupings. The components factored into separately declared records, based on a common quality or purpose, correspond to a lower level of abstraction than that represented by the larger record.

note

A carefully chosen name for the component of the larger record that is used to select the smaller enhances readability, for example:

```
if WINDOW1.BOTTOM_RIGHT.ROW > WINDOW2.TOP_LEFT.ROW then . . .
```

5.4.3 Dynamic Data

guideline

- Differentiate between static and dynamic data. Use dynamically allocated objects with caution.
- Use dynamically allocated data structures only when it is necessary to create and destroy them dynamically or to be able to reference them by different names.
- Do not drop pointers to undeallocated objects.
- Do not leave dangling references to deallocated objects.
- Initialize all access variables and components.
- Do not rely on memory deallocation.
- Deallocate explicitly.
- Use length clauses to specify total allocation size.
- Provide handlers for STORAGE_ERROR.

example

These lines show how a dangling reference might be created:

```
P1 := new OBJECT;
P2 := P1;
UNCHECKED_OBJECT_DEALLOCATION (P2);
```

This line can raise an exception due to referencing the deallocated object:

```
X := P1.DATA;
```

In the following three lines, if there is no intervening assignment of the value of P1 to any other pointer, the object created on the first line is no longer accessible after the third line. The only pointer to the allocated object has been dropped.

```
P1 := new OBJECT;
...
P1 := P2;
```

rationale

See also Guidelines 5.9.1, 5.9.2, and 6.1.3 for variations on these problems. A dynamically allocated object is an object created by the execution of an allocator ("new"). Allocated objects referenced by access variables allow you to generate aliases, which are multiple references to the same object. Anomalous behavior can arise when you reference a deallocated object by another name. This is called a dangling reference. Totally disassociating a still-valid object from all names is called dropping a pointer. A dynamically allocated object that is not associated with a name cannot be referenced or explicitly deallocated.

A dropped pointer depends on an implicit memory manager for reclamation of space. It also raises questions for the reader as to whether the loss of access to the object was intended or accidental.

An Ada environment is not required to provide deallocation of dynamically allocated objects. If provided, it may be provided implicitly (objects are deallocated when their access type goes out of scope), explicitly (objects are deallocated when UNCHECKED_DEALLOCATION is called), or both. To increase the likelihood of the storage space being reclaimed, it is best to call UNCHECKED_DEALLOCATION explicitly for each dynamically object when you are finished using it. Calls to UNCHECKED_DEALLOCATION also document a deliberate decision to abandon an object, making the code easier to read and understand. To be absolutely certain that space is reclaimed and reused, manage your own "free list." Keep track of which objects you are finished with, and reuse them instead of dynamically allocating new objects later.

The dangers of dangling references are that you may attempt to use them, thereby accessing memory which you have released to the memory manager, and which may have been subsequently allocated for another purpose in another part of your program. When you read from such memory, unexpected errors may occur because the other part of your program may have previously written totally unrelated data there. Even worse, when you write to such memory you can cause errors in an apparently unrelated part of the code by changing values of variables dynamically allocated by that code. This type of error can be very difficult to find. Finally, such errors may be triggered in parts of your environment that you didn't write, for example, in the memory management system itself which may dynamically allocate memory to keep records about your dynamically allocated memory.

Keep in mind that any uninitialized or unreset component of a record or array can also be a dangling reference or carry a bit pattern representing inconsistent data.

Whenever you use dynamic allocation it is possible to run out of space. Ada provides a facility (a length clause) for requesting the size of the pool of allocation space at compile time. Anticipate that you can still run out at run time. Prepare handlers for the exception STORAGE_ERROR, and consider carefully what alternatives you may be able to include in the program for each such situation.

There is a school of thought that dictates avoidance of all dynamic allocation. It is largely based on the fear of running out of memory during execution. Facilities such as length clauses and exception handlers for STORAGE_ERROR provide explicit control over memory partitioning and error recovery, making this fear unfounded.

5.5 EXPRESSIONS

Properly coded expressions can enhance the readability and understandability of a program. Poorly coded expressions can turn a program into a maintainer's nightmare.

5.5.1 Range Values

guideline

- Use 'FIRST or 'LAST instead of numeric literals to represent the first or last values of a range.

- Use the type or subtype name of the range instead of 'FIRST .. 'LAST.

example

```
subtype TEMPERATURE   is integer range ALL_TIME_LOW .. ALL_TIME_HIGH;
CURRENT_TEMPERATURE   :  TEMPERATURE;
type WEATHER_STATIONS is 1 .. MAX_STATIONS;
...
for I in WEATHER_STATIONS loop
   OFFSET := CURRENT_TEMPERATURE - TEMPERATURE'FIRST;
   ...
end loop;
```

rationale

In the example above, it is better to use WEATHER_STATIONS in the for loop than to use WEATHER_STATIONS'FIRST .. WEATHER_STATIONS'LAST or 1 .. MAX_STATIONS, because it is clearer, less error-prone, and less dependent on the definition of the type WEATHER_STATIONS. Similarly, it is better to use TEMPERATURE'FIRST in the offset calculation than to use ALL_TIME_LOW, because the code will still be correct if the definition of the subtype TEMPERATURE is changed. This enhances program reliability.

caution

When you implicitly specify ranges and attributes like this, be careful that you use the correct type or subtype name. It is easy to refer to a very large range without realizing it. For example, given the declarations:

```
type    LARGE_RANGE is new INTEGER;
subtype SMALL_RANGE is LARGE_RANGE range 1..10;
```

the first declaration below works fine, but the second one is probably an accident and raises an exception on most machines because it is requesting a huge array (indexed from the smallest integer to the largest one):

```
ARRAY_1 : array (SMALL_RANGE) of integer;
ARRAY_2 : array (LARGE_RANGE) of integer;
```

5.5.2 Array Attributes

guideline

- Use array attributes 'FIRST, 'LAST, or 'LENGTH instead of numeric literals for accessing arrays.
- Use the 'RANGE of the array instead of the name of the index type to express a range.
- Use 'RANGE instead of 'FIRST .. 'LAST to express a range.

example

```
subtype NAME_STRING_SIZE is POSITIVE range 1 .. 30;
NAME_STRING : STRING (NAME_STRING_SIZE);

for I in NAME_STRING'RANGE loop
   ...
end loop;
```

rationale

In the example above, it is better to use NAME_STRING'RANGE in the for loop than to use NAME_STRING_SIZE, NAME_STRING'FIRST .. NAME_STRING'LAST, or 1 .. 30, because it is clearer, less error-prone, and less dependent on the definitions of NAME_STRING and NAME_STRING_SIZE. If NAME_STRING is changed to have a different index type, or if the bounds of the array are changed, this will still work correctly. This enhances program reliability.

5.5.3 Parenthetical Expressions

guideline

- Use parentheses to specify the order of subexpression evaluation where operators from different precedence levels are involved, and to clarify expressions (General Dynamics 1986 and NASA 1987).

example

```
(1.5 * (X**2)) + (6.5 * X) + 47
```

rationale

Parenthetical expressions improve code readability. If you forget which operator has higher precedence, it may be helpful to use parentheses to specify the order of subexpression evaluation.

5.5.4 Positive Forms of Logic

guideline

- Avoid names and constructs that rely on the use of negatives.
- Choose names of flags so they represent states that can be used in positive form.

example

```
-- Use
if OPERATOR_MISSING
-- rather than either
if not OPERATOR_FOUND
-- or
if not OPERATOR_MISSING
```

rationale

Relational expressions can be more readable and understandable when stated in a positive form. As an aid in choosing the name, consider that the most frequently used branch in a conditional construct should be encountered first.

exception

There are cases in which the negative form is unavoidable. If the relational expression better reflects what is going on in the code, then inverting the test to adhere to this guideline is not recommended.

5.5.5 Short Circuit Forms of the Logical Operators

guideline

- Use short-circuit forms of the logical operators.

example

```
-- Use
if not (Y = 0) or else ((X / Y) /= 10) ...
-- or
if (Y /= 0) then
   if ((X / Y) / 10) then ...
-- rather than either
if (Y /= 0) and ((X / Y) = 10) ...
-- or
if ((X / Y) = 10) ...
-- to avoid NUMERIC_ERROR.

-- Use
if TARGET /= null and then TARGET.DISTANCE < THRESHOLD then ...
-- rather than
if TARGET.DISTANCE < THRESHOLD then ...
-- to avoid referencing a field in a non-existent object.
```

rationale

The use of short-circuit control forms prevents a class of data-dependent errors or exceptions that can occur as a result of expression evaluation. The short-circuit forms guarantee an order of evaluation and an exit from the sequence of relational expressions as soon as the expression's result can be determined.

In the absence of short-circuit forms, Ada does not provide a guarantee of the order of expression evaluation, nor does the language guarantee that evaluation of a relational expression is abandoned when it becomes clear that it evaluates to FALSE (for and) or TRUE (for or).

note

If it is important that all parts of a given expression always be evaluated, the expression probably violates Guideline 4.1.3 which prohibits side-effects in functions.

5.5.6 Type Qualified Expressions and Type Conversions

guideline

- Use type qualified expressions instead of type conversions wherever possible.

example

```
type REAL is ...
type WHOLE is ...

ACTUAL_SPEED  : REAL;
DESIRED_SPEED : WHOLE; -- Console dial setting
TAIL_WIND     : WHOLE; -- Cheap sensor

ACTUAL_SPEED := REAL (DESIRED_SPEED + TAIL_WIND);
-- A type conversion. An addition operation inherited by subtype WHOLE is
--  used, followed by conversion of the result to REAL.

ACTUAL_SPEED := REAL'(DESIRED_SPEED + TAIL_WIND);
-- A type qualified expression. A specific operator overloading + and
--  giving result type REAL is used.
```

rationale

Type qualified expressions are evaluated at compile time, but type conversions are made at execution time. Type qualifiers help in operator overload resolution by explicitly specifying the qualified expressions' desired result type.

5.5.7 Accuracy of Operations With Real Operands

guideline

- Use <= and >= in relational expressions with real operands instead of =.

example

```
CURRENT_TEMPERATURE    : TEMPERATURE := 0.0;
TEMPERATURE_INCREMENT  : TEMPERATURE := 1.0 / 3.0;
MAXIMUM_TEMPERATURE    : constant    := 100.0;

...
loop
    ...
    CURRENT_TEMPERATURE := CURRENT_TEMPERATURE + TEMPERATURE_INCREMENT;
    ...
    exit when CURRENT_TEMPERATURE >= MAXIMUM_TEMPERATURE;
    ...
end loop;
```

rationale

Fixed and floating point values, even if derived from similar expressions, may not be exactly equal. The imprecise, finite representations of real numbers in hardware always have round-off errors so that any variation in the construction path or history of two reals has the potential for resulting in different numbers, even when the paths or histories are mathematically equivalent.

The Ada definition of model intervals also means that the use of <= is more transportable than either < or =.

note

Floating point arithmetic is treated in Chapter 7.

exceptions

If your application must test for an exact value of a real number (e.g., testing the precision of the arithmetic on a certain machine), then the = would have to be used. But never use = on real operands as a condition to exit a loop.

5.6 STATEMENTS

Careless or convoluted use of statements can make a program hard to read and maintain even if its global structure is well organized. You should strive for simple and consistent use of statements to achieve clarity of local program structure. Some of the guidelines in this section counsel use or avoidance of particular statements. As pointed out in the individual guidelines, rigid adherence to those guidelines would be excessive, but experience has shown that they generally lead to code with improved reliability and maintainability.

5.6.1 Nesting

guideline

- Minimize the depth of nested expressions (Nissen and Wallis 1984).
- Minimize the depth of nested control structures (Nissen and Wallis 1984).
- Try simplification heuristics.

instantiation

- Do not nest expressions or control structures beyond a nesting level of five.

example

The following section of code:

```
if not CONDITION_1 then
   if CONDITION_2 then
      ACTION_A;
   else
      ACTION_B;
   end if;
else
   ACTION_C;
end if;
```

can be rewritten more clearly and with less nesting as:

```
if CONDITION_1 then
   ACTION_C;
elsif CONDITION_2 then
   ACTION_A;
else
   ACTION_B;
end if;
```

rationale

Deeply nested structures are confusing, difficult to understand, and difficult to maintain. The problem lies in the difficulty of determining what part of a program is contained at any given level. For expressions, this is important in achieving the correct placement of balanced grouping symbols and in achieving the desired operator precedence. For control structures, the question involves what part is controlled. Specifically, is a given statement at the proper level of nesting, i.e., is it too deeply or too shallowly nested, or is the given statement associated with the proper choice, e.g., for if or case statements? Indentation helps, but it is not a panacea. Visually inspecting alignment of indented code (mainly intermediate levels) is an uncertain job at best. To minimize the complexity of the code, keep the maximum number of nesting levels between three and five.

note

Ask yourself the following questions to help you consider alternatives to the code and help you reduce the nesting:

- Can some part of the expression be put into a constant or variable?
- Does some part of the lower nested control structures represent a significant, and perhaps reusable computation that I can factor into a subprogram?
- Can I convert these nested if statements into a case statement?
- Am I using `else if` where I could be using `elsif`?
- Can I reorder the conditional expressions controlling this nested structure?
- Is there a different design that would be simpler?

exceptions

If deep nesting is required frequently, there may be overall design decisions for the code that should be changed. Some algorithms require deeply nested loops and segments controlled by conditional branches. Their continued use can be ascribed to their efficiency, familiarity, and time proven utility. When nesting is required, proceed cautiously and take special care with the choice of identifiers and loop and block names.

5.6.2 Slices

guideline

- Use slices rather than a loop to copy part of an array.

example

```
type SQUARE_MATRIX   is array (ROWS, ROWS) of ELEMENT;
type DIAGONALS       is array (1 .. 3)     of ELEMENT;
type ROW_VECTOR      is array (ROWS)       of ELEMENT;
type TRI_DIAGONAL    is array (ROWS)       of DIAGONALS;
MARKOV_PROBABILITIES :  SQUARE_MATRIX;
DIAGONAL_DATA        :  TRI_DIAGONAL;
...
...
-- Remove diagonal and off-diagonal elements.
DIAGONAL_DATA (ROWS'FIRST) (1)       := NULL_VALUE;
DIAGONAL_DATA (ROWS'FIRST) (2 .. 3) :=
  MARKOV_PROBABILITIES (ROWS'FIRST) (ROWS'FIRST .. ROWS'SUCC (ROWS'FIRST));

for I in ROWS'SUCC (ROWS'FIRST) .. ROWS'PRED (ROWS'LAST) loop
   DIAGONAL_DATA (I) := MARKOV_PROBABILITIES (I) (I - 1 .. I + 1);
end loop;

DIAGONAL_DATA (ROWS'LAST) (1 .. 2) :=
  MARKOV_PROBABILITIES (ROWS'LAST) (ROWS'PRED (ROWS'LAST) .. ROWS'LAST);
DIAGONAL_DATA (ROWS'LAST) (3)       := NULL_VALUE;
```

rationale

An assignment statement with slices is simpler and clearer than a loop, and helps the reader see the intended action. Slice assignment can be faster than a loop if a block move instruction is available.

5.6.3 Case Statements

guideline

- Never use an `others` choice in a case statement.
- Do not use ranges of enumeration literals in case statements.
- If you use an if statement instead of a case statement, use marker comments indicating the cases.

example

```
type COLOR is (RED, GREEN, BLUE, PURPLE);

case ...
   when RED .. BLUE => ...
   when PURPLE      => ...
end case;
```

Now consider a change in the type:

```
type COLOR is (RED, YELLOW, GREEN, BLUE, PURPLE);
```

This change may have an unnoticed and undesired effect in the case statement. If the choices had been enumerated explicitly, as `when RED, GREEN, BLUE =>` instead of `when RED .. BLUE =>`, then the case statement would have not have compiled. This would have forced the maintainer to make a conscious decision about what to do in the case of `YELLOW`.

rationale

All possible values for an object should be known and should be assigned specific actions. Use of an others clause may prevent the developer from carefully considering the actions for each value. A compiler warns the user about omitted values, if an others clause is not used.

Each possible value should be explicitly enumerated. Ranges can be dangerous because of the possibility that the range could change and the case statement may not be reexamined.

A case statement can be more efficient than a nested if-then-else structure. Where the case statement is less efficient, marking the if statement documents the intended purpose and allows the if to be converted back to a case should the code move to a different implementation or machine.

exception

It is acceptable to use ranges for possible values only when the user is certain that new values will never be inserted among the old ones, as for example, in the range of ASCII characters: `'a' .. 'z'`.

5.6.4 Loops

guideline

- Use for loops whenever possible (when the number of iterations is computable).
- Use while loops when the number of iterations is not computable, but a simple continuation condition can be applied at the top of the loop.
- Use plain loops with exit statements for more complex situations.
- Avoid exit statements in while and for loops.
- Minimize the number of ways to exit a loop.

example

To iterate over all elements of an array:

```
for I in ARRAY_NAME'RANGE loop
   ...
end loop;
```

To iterate over all elements in a linked list:

```
POINTER := HEAD_OF_LIST;
while (POINTER /= null) loop
   ...
   POINTER := POINTER.NEXT;
end loop;
```

Situations requiring a "loop and a half" arise often. For this use:

```
P_AND_Q_PROCESSING:
loop
   P;
   exit P_AND_Q_PROCESSING when CONDITION_DEPENDENT_ON_P;
   Q;
end loop;
```

rather than:

```
P;
while not CONDITION_DEPENDENT_ON_P loop
   Q;
   P;
end loop;
```

rationale

A for loop is bounded, so it cannot be an "infinite loop." This is enforced by the Ada language which requires a finite range in the loop specification and which does not allow the loop counter of a for loop to be modified by a statement executed within the loop. This yields a certainty of understanding for the reader and the writer not associated with other forms of loops. A for loop is also easier to maintain because the iteration range can be expressed using attributes of the data structures upon which the loop operates, as shown in the example above where the range changes automatically whenever the declaration of the array is modified. For these reasons, it is best to use the for loop whenever possible; that is, whenever simple expressions can be used to describe the first and last values of the loop counter.

The while loop has become a very familiar construct to most programmers. At a glance it indicates the condition under which the loop continues. Use the while loop whenever it is not possible to use the for loop, but there is a simple boolean expression describing the conditions under which the loop should continue, as shown in the example above.

The normal loop statement should be used in more complex situations, even if it is possible to contrive a solution using a for or while loop in conjunction with extra flag variables or exit statements. The criteria in selecting a loop construct is to be as clear and maintainable as possible. It is a bad idea to use an exit statement from within a for or while loop because it is misleading to the reader after having apparently described the complete set of loop conditions at the top of the loop. A reader who encounters a normal loop statement expects to see exit statements.

There are some familiar looping situations which are best achieved with the normal loop statement. For example, the semantics of the Pascal repeat until loop, where the loop is always executed at least once before the termination test occurs, are best achieved by a normal loop with a single exit at the end of the loop. Another common situation is the "loop and a half" construct, shown in the example above, where a loop must terminate somewhere within the sequence of statements of the body. Complicated "loop and a half" constructs simulated with while loops often require the introduction of flag variables, or duplication of code before and during the loop, as shown in the example. Such contortions make the code more complex and less reliable.

Minimize the number of ways to exit a loop in order to make the loop more understandable to the reader. It should be rare that you need more than two exit paths from a loop. When you do, be sure to use exit statements for all of them, rather than adding an exit statement to a for or while loop.

5.6.5 Exit Statements

guideline

- Use exit statements to enhance the readability of loop termination code (NASA 1987).
- Use `exit when ...` rather than `if ... then exit` whenever possible (NASA 1987).
- Review exit statement placement.

example

See the examples in Guidelines 5.1.1 and 5.6.4.

rationale

It is more readable to use exit statements than to try to add boolean flags to a while loop condition to simulate exits from the middle of a loop. Even if all exit statements would be clustered at the top of the loop body, the separation of a complex condition into multiple exit statements can simplify and make it more readable and clear. The sequential execution of two exit statements is often more clear than the short-circuit control forms.

The `exit when` form is preferable to the `if ... then, exit` form because it makes the word `exit` more visible by not nesting it inside of any control construct. The `if ... then exit` form is needed only in the

case where other statements, in addition to the exit statement, must be executed conditionally. For example:

```
if STATUS = DONE then
  SHUT_DOWN;
   exit;
end if;
```

Loops with many scattered exit statements can indicate fuzzy thinking as regards the loop's purpose in the algorithm. Such an algorithm might be coded better some other way, e.g., with a series of loops. Some rework can often reduce the number of exit statements and make the code clearer.

See also Guidelines 5.1.3 and 5.6.4.

5.6.6 Recursion and Interaction Bounds

guideline

- Understand and consider specifying bounds on loops.
- Understand and consider specifying bounds on recursion.

example

Establishing an iteration bound:

```
SAFETY_COUNTER := 0;
PROCESS_LIST:
   loop
      exit when CURRENT_ITEM = null;
      ...
      CURRENT_ITEM := CURRENT_ITEM.NEXT;
      ...
      SAFETY_COUNTER := SAFETY_COUNTER + 1;
      if SAFETY_COUNTER > 1_000_000 then
         raise SAFETY_ERROR;
      end if;
   end loop PROCESS_LIST;
```

Establishing a recursion bound:

```
-------------------------------------------------------------------------
procedure DEPTH_FIRST (ROOT           : in SUBTREE;
                       ...
                       SAFETY_COUNTER : in RECURSION_BOUND := 1_000)    is
begin
   if SAFETY_COUNTER = 0 then
      raise RECURSION_ERROR;
   end if;
   ... -- normal subprogram body
   DEPTH_FIRST (SUB_ROOT, ..., (SAFETY_COUNTER - 1)); -- recursive    call
   ...
end DEPTH_FIRST;
-------------------------------------------------------------------------
```

Following are examples of this subprogram's usage. One call specifies a maximum recursion depth of 50. The second takes the default (one thousand). The third uses a computed bound:

```
DEPTH_FIRST (TREE, ..., 50);
DEPTH_FIRST (TREE, ... );
DEPTH_FIRST (TREE, ..., CURRENT_TREE_HEIGHT);
```

rationale

Recursion, and iteration using structures other than for statements, can be infinite because the expected terminating condition does not arise. Such faults are sometimes quite subtle, may occur rarely, and may be difficult to detect because an external manifestation might be absent or substantially delayed.

By including counters and checks on the counter values, in addition to the loops themselves, you can prevent many forms of infinite loops. The inclusion of such checks is one aspect of the technique called Safe Programming (Anderson and Witty 1978).

The bounds of these checks do not have to be exact, just realistic. Such counters and checks are not part of the primary control structure of the program but a benign addition functioning as an execution-time "safety net" allowing error detection and possibly recovery from potential infinite loops or infinite recursion.

note

If a loop uses the `for` iteration scheme (Guideline 5.6.4), it follows this guideline.

exceptions

Embedded control applications have loops that are intended to be infinite. Only a few loops within such applications should qualify as exceptions to this guideline. The exceptions should be deliberate (and documented) policy decisions.

This guideline is most important to safety critical systems. For other systems, it may be overkill.

5.6.7 Goto Statements

guideline

- Do not use goto statements unless you are sure there is no alternative.
- If you must use a goto statement, highlight both it and the label.

rationale

A goto statement is an unstructured change in the control flow. Worse, the label does not require an indicator of where the corresponding goto statement(s) are. This makes code unreadable and makes its correct execution suspect.

note

For the rare occasions in which you can present a case for using a goto statement, highlight both it and the label with blank space and highlighting comments, and indicate at the label where the corresponding goto statement(s) may be found.

5.6.8 Return Statements

guideline

- Minimize the number of returns from a subprogram (NASA 1987).
- Highlight returns with comments or white space to keep them from being lost in other code.

example

The following code fragment is longer and more complex than necessary:

```
if (POINTER /= null) then
   if (POINTER.COUNT > 0) then
      return TRUE;
   else
      return FALSE;
   end if;
else
   return FALSE;
end if;
```

It should be replaced with the shorter, more concise, and clearer equivalent line:

```
return (POINTER /= null and then POINTER.COUNT > 0);
```

rationale

Excessive use of returns can make code confusing and unreadable. Only use return statements where warranted. Too many returns from a subprogram may be an indicator of cluttered logic. If the application requires multiple returns, use them at the same level (i.e., as in different branches of a case statement), rather than scattered throughout the subprogram code. Some rework can often reduce the number of return statements to one and make the code more clear.

exception

Do not avoid return statements if it detracts from natural structure and code readability.

5.6.9 Blocks

guideline

- Use blocks liberally and for their intended purposes.

example

```
...
INTEGRATE_VELOCITY_FROM_ACCELERATION:
   begin
      ...
   exception
      when NUMERIC_ERROR | CONSTRAINT_ERROR =>
         ... -- use old velocity value
   end INTEGRATE_VELOCITY_FROM_ACCELERATION;
...
```

rationale

Blocks break up large segments of code and isolate details relevant to each subsection of code. The intended purposes of blocks are to introduce local declarations, define local exception handlers, and perform local renaming.

Declaring objects locally limits the visibility to the scope of the block. This enforces information hiding. It also allows for memory deallocation when the block is done executing. Local exception handlers can catch exceptions close to the point of origin and allow them to either be handled, propogated, or converted. Local renaming enhances readability for a given section of code.

5.6.10 Aggregates

guideline

- Use an aggregate instead of a sequence of assignments to assign values to all fields of a record.
- Use an aggregate instead of a temporary variable when building a record to pass as an actual parameter.
- Use named component association in aggregates.

example

It is better to use aggregates:

```
SET_POSITION ((X => 100, Y => 200));

EMPLOYEE_RECORD :=
  (NUMBER     => 42;
   AGE        => 51;
   DEPARTMENT => SOFTWARE_ENGINEERING);
```

than to use consecutive assignments or temporary variables:

```
TEMPORARY_POSITION : POSITION;
...
TEMPORARY_POSITION.X := 100;
TEMPORARY_POSITION.Y := 200;
SET_POSITION (TEMPORARY_POSITION);

EMPLOYEE_RECORD.NUMBER     := 42;
EMPLOYEE_RECORD.AGE        := 51;
EMPLOYEE_RECORD.DEPARTMENT := SOFTWARE_ENGINEERING;
```

rationale

Use of aggregates is beneficial during maintenance. If a record structure is altered, but the corresponding aggregate is not, the compiler flags the missing field in the aggregate assignment. It would not be able to detect the fact that a new assignment statement should have been added to a list of assignment statements.

Aggregates can also be a real convenience in combining data items into a record or array structure required for passing the information as a parameter. Named component association makes aggregates more readable.

5.7 VISIBILITY

As noted in Section 4.2, Ada's ability to enforce information hiding and separation of concerns through its visibility controlling features is one of the most important advantages of the language. Subverting these features, for example by over liberal use of the use clause, is wasteful and dangerous.

5.7.1 The Use Clause

guideline

- Minimize using the use clause (Nissen and Wallis 1984).
- Localize the effect of the use clauses you must employ.

example

This is a modification of the example from Guideline 4.2.3. The effect of a use clause is localized.

```
-------------------------------------------------------------------------
procedure COMPILER is
   ----------------------------------------------------------------------
   package LISTING_FACILITIES is

      procedure NEW_PAGE_OF_LISTING;
      procedure NEW_LINE_OF_PRINT;
      -- etc.

   end LISTING_FACILITIES;
   ----------------------------------------------------------------------
   package body LISTING_FACILITIES  is separate;
   ----------------------------------------------------------------------
begin --COMPILER
   ...
end COMPILER;
-------------------------------------------------------------------------


with TEXT_IO;
   separate (COMPILER)
package body LISTING_FACILITIES is
   ----------------------------------------------------------------------
   procedure NEW_PAGE_OF_LISTING is
   begin
      ...
   end NEW_PAGE_OF_LISTING;
   ----------------------------------------------------------------------
   procedure NEW_LINE_OF_PRINT is
      use TEXT_IO;                                  -- Note use clause.
   begin
      ...
   end NEW_LINE_OF_PRINT;
   ----------------------------------------------------------------------
   -- etc

end LISTING_FACILITIES;
-------------------------------------------------------------------------
```

rationale

Avoiding the use clause forces you to use fully qualified names. In large systems, there may be many library units named in with clauses. When corresponding use clauses accompany the with clauses, and the simple names of the library packages are omitted (as is allowed by the use clause), references to external entities are obscured, and identification of external dependencies becomes difficult.

You can minimize the scope of the use clause by placing it in the body of a package or subprogram, or encapsulating it in a block to restrict visibility. Placing a use clause in a block has a similar effect to the Pascal with statement of localizing the use of unqualified names.

notes

Avoiding the use clause completely can cause problems when compiling with (in the context of) packages that contain type declarations. Simply importing these types via a with clause does not allow relational operators implicitly defined for them to be used in infix notation. A use clause enables the use of infix notation. A better choice is to use renaming declarations to overcome the visibility problem and enable the use of infix notation.

Avoiding the use clause completely also causes problems with enumeration literals, which must then be fully qualified. This problem can be solved by declaring constants with the enumeration literals as their values, except that such constants cannot be overloaded like enumeration literals.

An argument defending the use clause can be found in (Rosen 1987).

automation note

There are tools which can analyze your Ada source code, resolving overloading of names, and automatically converting to the use or avoidance of use clauses.

5.7.2 The Renames Clause

guideline

- Use the renames clause judiciously and purposefully.
- Rename a long fully qualified name to reduce the complexity if it becomes unwieldy (Guideline 3.1.4).
- Rename declarations for visibility purposes rather than using the use clause especially infix operators (Guideline 5.7.1).
- Rename parts when interfacing to reusable components originally written with nondescriptive or inapplicable nomenclature.
- Use a project-wide standard list of abbreviations to rename common packages.

example

```
procedure DISK_WRITE (TRACK_NAME : in TRACK; ITEM : in DATA) renames
   SYSTEM_SPECIFIC.DEVICE_DRIVERS.DISK_HEAD_SCHEDULER.TRANSMIT;
```

rationale

If the renaming facility is abused, the code can be difficult to read. A renames clause can substitute an abbreviation for a qualifier or long package name locally. This can make code more readable yet anchor the code to the full name. However, the use of renames clauses can often be avoided or made obviously undesirable by choosing names carefully so that fully qualified names read well. The list of renaming declarations serves as a list of abbreviation definitions (see Guideline 3.1.4). By renaming imported infix operators, the use clause can often be avoided. The method prescribed in the Ada Language Reference Manual (Department of Defense 1983) for renaming a type is to use a subtype (see Guideline 3.4.1). Often the parts recalled from a reuse library do not have names that are as general as they could be or that match the new application's naming scheme. An interface package exporting the renamed subprograms can map to your project's nomenclature.

5.7.3 Overloaded Subprograms

guideline

- Limit overloading to widely used subprograms that perform similar actions on arguments of different types (Nissen and Wallis 1984).

example

```
function SIN (ANGLES : MATRIX_OF_RADIANS) return MATRIX;
function SIN (ANGLES : VECTOR_OF_RADIANS) return VECTOR;
function SIN (ANGLE  : RADIANS           ) return SMALL_REAL;
function SIN (ANGLE  : DEGREES           ) return SMALL_REAL;
```

rationale

Excessive overloading can be confusing to maintainers (Nissen and Wallis 1984, 65). Only use it when there is an overwhelming reason to do so. There is also the danger of hiding declarations if overloading becomes habitual.

note

This guideline does not prohibit subprograms with identical names declared in different packages.

5.7.4 Overloaded Operators

guideline

- Preserve the conventional meaning of overloaded operators (Nissen and Wallis 1984).
- Use "+" to identify adding, joining, increasing, and enhancing kinds of functions.
- Use "-" to identify subtraction, separation, decreasing, and depleting kinds of functions.

example

```
function "+" (X : MATRIX;
              Y : MATRIX) return MATRIX;
...
SUM_MATRIX := MATRIX_A + MATRIX_B;
```

rationale

Subverting the conventional interpretation of operators leads to confusing code.

note

There are potential problems with any overloading. For example, if there are several versions of the "+" operator, and a change to one of them affects the number or order of its parameters, locating the occurrences that must be changed can be difficult.

5.8 USING EXCEPTIONS

Ada exceptions are a reliability-enhancing language feature designed to help specify program behavior in the presence of errors or unexpected events. Exceptions are not intended to provide a general purpose control construct. Further, liberal use of exceptions should not be considered sufficient for providing full software fault tolerance (Melliar-Smith and Randall 1987).

This section addresses the issues of how and when to avoid raising exceptions, how and where to handle them, and whether to propagate them. Information on how to use exceptions as part of the interface to a unit, including what exceptions to declare and raise and under what conditions to raise them.

5.8.1 Handling Versus Avoiding Exceptions

guideline

- Avoid causing exceptions to be raised when it is easy and efficient to do so.
- Provide handlers for exceptions which cannot be avoided.
- Use exception handlers to enhance readability by separating fault handling from normal execution.
- Do not use exceptions and exception handlers as goto statements.

rationale

In many cases, it is possible to detect easily and efficiently that an operation you are about to perform would raise an exception. In such a case, it is a good idea to do so rather than allowing the exception to be raised handling it with an exception handler. For example, check each pointer for NULL when traversing a linked list of records connected by pointers. Also, test an integer for zero before dividing by it, and call an interrogative function STACK_IS_EMPTY before invoking the POP procedure of a stack package. Such tests are appropriate when they can be performed easily and efficiently, as a natural part of the algorithm being implemented.

However, error detection in advance is not always so simple. There are cases where such a test is too expensive or too unreliable. In such cases, it is better to attempt the operation within the scope of an exception handler so that the exception is handled if it is raised. For example, in the case of a linked list implementation of a list, it is very inefficient to call a function ENTRY_EXISTS before each call to the procedure MODIFY_ENTRY simply to avoid raising the exception ENTRY_NOT_FOUND. It takes as much time to search the list to avoid the exception as it takes to search the list to perform the update. Similarly, it is much easier to attempt a division by a real number within the scope of an exception handler to handle numeric overflow than to test in advance whether the dividend is too large or the divisor too small for the quotient to be representable on the machine.

In concurrent situations, tests done in advance can also be unreliable. For example, if you want to modify an existing file on a multi-user system, it is safer to attempt to do so within the scope of an exception handler than to test in advance whether the file exists, whether it is protected, whether there is room in the file system for the file to be enlarged, etc. Even if you tested for all possible errors conditions, there is no guarantee that nothing would change after the test and before the modification operation. You still need the exception handlers, so the advance testing serves no purpose.

Whenever such a case does not apply, normal and predictable events should be handled by the code without the abnormal transfer of control represented by an exception. When fault handling and only fault handling code is included in exception handlers, the separation makes the code easier to read. The reader can skip all the exception handlers and still understand the normal flow of control of the code. For this reason, exceptions should never be raised and handled within the same unit, as a form of a goto statement to exit from a loop, if, case, or block statement.

5.8.2 Handlers for others

guideline

- Use caution when programming handlers for `others`.
- Provide a handler for `others` in suitable frames to protect against unexpected exceptions being propagated without bound, especially in safety critical systems.
- Use `others` only to catch exceptions you cannot enumerate explicitly, preferably only to flag a potential abort.
- Avoid using `others` during development.

rationale

Providing a handler for `others` allows you to follow the other guidelines in this section. It affords a place to catch and convert truly unexpected exceptions that were not caught by the explicit handlers. While it may be possible to provide "fire walls" against unexpected exceptions being propagated without providing handlers in every block, you can convert the unexpected exceptions as soon as they arise. The `others` handler cannot discriminate between different exceptions, and, as a result, any such handler must treat the exception as a disaster. Even such a disaster can still be converted into a user-defined exception at that point. Since a handler for `others` catches any exception not otherwise handled explicitly, one placed in the frame of a task or of the main subprogram affords the opportunity to perform final clean-up and to shut down cleanly.

Programming a handler for `others` requires caution because it cannot discriminate either which exception was actually raised or precisely where it was raised. Thus, the handler cannot make any assumptions about what can be or even what needs to be "fixed."

The use of handlers for `others` during development, when exception occurrences can be expected to be frequent, can hinder debugging. It is much more informative to the developer to see a traceback with the actual exception listed than the converted exception. Furthermore, many tracebacks do not list the point where the original exception was raised if it was caught by a handler.

note

The arguments in the preceding paragraph apply only to development time, when traceback listings are useful. They are not useful to users and can be dangerous. The handler should be included in comment form at the outset of development and the double dash removed before delivery.

5.8.3 Propagation

guideline

- Handle all exceptions, both user and predefined.
- For every exception that might be raised, provide a handler in suitable frames to protect against undesired propagation outside the abstraction.

rationale

The statement that "it can never happen" is not an acceptable programming approach. You must assume it can happen and be in control when it does. You should provide defensive code routines for the "cannot get here" conditions.

Some existing advice calls for catching and propagating any exception to the calling unit. This advice can stop a program. You should catch the exception and propagate it, or a substitute, only if your handler is at the wrong abstraction level to effect recovery. Effecting recovery can be difficult, but the alternative is a program that does not meet its specification.

Making an explicit request for termination implies that your code is in control of the situation and has determined that to be the only safe course of action. Being in control affords opportunities to shut down in a controlled manner (clean up loose ends, close files, release surfaces to manual control, sound alarms), and implies that all available programmed attempts at recovery have been made.

5.8.4 Localizing the Cause of an Exception

guideline

- Do not rely on being able to identify the fault raising predefined or implementation-defined exceptions.
- Use blocks to associate localized sections of code with their own exception handlers.

example

See Guideline 5.6.9.

rationale

It is very difficult to determine in an exception handler exactly which statement and which operation within that statement raised an exception, particularly the predefined and implementation-defined exceptions. The predefined and implementation-defined exceptions are candidates for conversion and propagation to higher abstraction levels for handling there. User-defined exceptions, being more closely associated with the application, are better candidates for recovery within handlers.

User-defined exceptions can also be difficult to localize. Associating handlers with small blocks of code helps to narrow the possibilities, making it easier to program recovery actions. The placement of handlers in small blocks within a subprogram or task body also allows resumption of the subprogram or task after the recovery actions. If you do not handle exceptions within blocks, the only action available to the handlers is to shut down the task or subprogram as prescribed in Guideline 5.8.3.

note

The optimal size for the sections of code you choose to protect by a block and its exception handlers is very application-dependent. Too small a granularity forces you to expend much more effort in programming for abnormal actions than for the normal algorithm. Too large a granularity reintroduces the problems of determining what went wrong and of resuming normal flow.

5.9 ERRONEOUS EXECUTION

An Ada program is erroneous when it violates or extends the rules of the language governing program behavior. Neither compilers nor run-time environments are able to detect erroneous behavior in all circumstances and contexts. The effects of erroneous execution are unpredictable (Ada Reference Manual 1983, §1.6). If the compiler does detect an instance of an erroneous program, its options are to indicate a

compile time error, to insert the code to raise PROGRAM_ERROR, and possibly to write a message to that effect, or to do nothing at all.

Erroneousness is not a concept unique to Ada. The guidelines below describe or explain the specific instances of erroneousness defined in the Ada Language Reference Manual (Department of Defense 1983).

5.9.1 Unchecked Conversion

guideline

- Use UNCHECKED_CONVERSION only with the utmost care (Ada Reference Manual 1983, §13.10.2).
- Isolate the use of UNCHECKED_CONVERSION in package bodies.

rationale

An unchecked conversion is a bit-for-bit copy without regard to the meanings attached to those bits and bit positions by either the source or the destination type. The source bit pattern can easily be meaningless in the context of the destination type. Unchecked conversions can create values that violate type constraints on subsequent operations. Unchecked conversion of objects mismatched in size has implementation-dependent results.

5.9.2 Unchecked Deallocation

guideline

- Use UNCHECKED_DEALLOCATION purposefully and carefully.
- Isolate the use of UNCHECKED_DEALLOCATION in package bodies.

rationale

Most of the reasons for using unchecked deallocation with caution have been given in Guideline 5.4.3. When this feature is used, there is no checking that there is only one access path to the storage being deallocated. Thus, any other access paths are not made null. Depending on such a check is erroneous.

5.9.3 Dependence on Parameter Passing Mechanism

guideline

- Do not write code whose correct execution depends on the particular parameter passing mechanism used by an implementation (Ada Reference Manual 1983 and Cohen 1986).

example

The output of this program depends on the particular parameter passing mechanism that was used.

```
-------------------------------------------------------------------------
with TEXT_IO;
use  TEXT_IO;
procedure OUTER is
   type COORDINATES is
      record
         X : INTEGER := 0;
         Y : INTEGER := 0;
      end record;

   OUTER_POINT : COORDINATES;

   package INTEGER_IO is new TEXT_IO.INTEGER_IO (INTEGER);
   use INTEGER_IO;
   -----------------------------------------------------------------------
   procedure INNER (INNER_POINT : in out COORDINATES) is
   begin -- INNER
      INNER_POINT.X := 5;

      -- The following line causes the output of the program to
      -- depend on the parameter passing mechanism.
      PUT (OUTER_POINT.X);
   end INNER;
   -----------------------------------------------------------------------
begin -- OUTER
   PUT (OUTER_POINT.X);
   INNER (OUTER_POINT);
   PUT (OUTER_POINT.X);
end OUTER;
-------------------------------------------------------------------------
```

If the parameter passing mechanism is by copy, the results on the standard output file are:

```
0 0 5
```

If the parameter passing mechanism is by reference, the results are:

```
0 5 5
```

rationale

The language definition specifies that a parameter whose type is an array, record, or task type can be passed by copy or reference. It is erroneous to assume that either mechanism is used in a particular case.

exceptions

Frequently, when interfacing Ada to foreign code, dependence on parameter passing mechanisms used by a particular implementation is unavoidable. In this case, isolate the calls to the foreign code in an interface package that exports operations that do not depend on the parameter-passing mechanism.

5.9.4 Multiple Address Clauses

guideline

- Use address clauses to map variables and entries to the hardware device or memory, not to model the FORTRAN "equivalence" feature.

example

```
SINGLE_ADDRESS : constant := ...
...

INTERRUPT_VECTOR_TABLE : HARDWARE_ARRAY;
   for INTERRUPT_VECTOR_TABLE use at SINGLE_ADDRESS;
```

rationale

The result of specifying a single address for multiple objects or program units is undefined, as is specifying multiple addresses for a single object or program unit. Specifying multiple address clauses for an interrupt entry is also undefined. It does not necessarily overlay objects or program units, or associate a single entry with more than one interrupt.

5.9.5 Suppression of Exception Check

guideline

- Do not suppress exception checks during development.
- Minimize suppression of exception checks during operation.

rationale

If you disable exception checks and program execution results in a condition in which an exception would otherwise occur, the program execution is erroneous. The results are unpredictable. Further, you must still be prepared to deal with the suppressed exceptions if they are raised in and propagated from the bodies of subprograms, tasks, and packages you call.

The pragma SUPPRESS grants an implementation permission to suppress run time checks, but it does not require it to do so. It cannot be relied upon as a general technique for performance improvement.

If you need to use pragma SUPPRESS, postpone it until it is clear that the program is correct, but too slow, and there is no other alternative for improving performance. Pragma SUPPRESS can then be used to improve performance for specific, well-understood objects/types.

5.9.6 Initialization

guideline

- Initialize all objects prior to use.
- Ensure elaboration of an entity before using it.
- Use function calls in declarations cautiously.

example

```
-------------------------------------------------------------------------
package ROBOT_CONTROLLER is
   ...
   function SENSE return POSITION;
   ...
end ROBOT_CONTROLLER;
-------------------------------------------------------------------------
package body ROBOT_CONTROLLER is
   ...
   GOAL : POSITION := SENSE;
                      --------              The underlined text is illegal.
   ...
   -------------------------------------------------------------------------
   function SENSE return POSITION is
      ...
   end SENSE;
   -------------------------------------------------------------------------
begin -- ROBOT_CONTROLLER
   GOAL := SENSE;                      -- This line is legal.
   ...
end ROBOT_CONTROLLER;
-------------------------------------------------------------------------
```

rationale

Ada does not define an initial default value for objects of any type other than access types. Using the value of an object before it has been assigned a value causes unpredictable behavior, possibly raising an exception. Objects can be initialized implicitly by declaration or explicitly by assignment statements. Initialization at the point of declaration is safest as well as easiest for maintainers. You can also specify default values for fields of records as part of the type declarations for those records.

Ensuring initialization does not imply initialization at the declaration. In the example above, GOAL must be initialized via a function call. This cannot occur at the declaration, because the function SENSE has not yet been elaborated, but can occur later as part of the sequence of statements of the body of the enclosing package.

An unelaborated function called within a declaration (initialization) raises an exception that must be handled outside of the unit containing the declarations. This is true for any exception the function raises even if it has been elaborated.

If an exception is raised by a function call in a declaration, it is not handled in that immediate scope. It is raised to the enclosing scope. This can be controlled by nesting blocks.

note

Sometimes, elaboration order can be dictated with pragma ELABORATE. Pragma ELABORATE only applies to library units.

5.10 SUMMARY

optional parts of the syntax

- Associate names with loops when they are nested.
- Associate names with blocks when they are nested.
- Use loop names on all exit statements.
- Include the simple name at the end of a package specification and body.
- Include the simple name at the end of a task specification and body.
- Include the simple name at the end of an accept statement.
- Include the designator at the end of a subprogram body.

parameter lists

- Name formal parameters descriptively to reduce the need for comments.
- Use named parameter association in calls of infrequently used subprograms or entries with many formal parameters.
- Use named association for constants, expressions, and literals in aggregates.
- Use named association when instantiating generics.
- Use named association for clarification when the actual parameter is any literal or expression.
- Use named association when supplying a nondefault value to an optional parameter.
- Provide default parameters to allow for occasional, special use of widely used subprograms or entries.
- Place default parameters at the end of the formal parameter list.
- Consider providing default values to new parameters added to an existing subprogram.
- Show the mode indication of all procedure and entry parameters.
- Select the most restrictive mode possible.
- Declare parameters in a consistent order.

types

- Use existing types as building blocks by deriving new types from them.
- Use range constraints on subtypes.
- Define new types, especially derived types, to include the largest set of possible values, including boundary values.
- Constrain the ranges of derived types with subtypes, excluding boundary values.
- Do not use anonymous types.
- Use limited private types in preference to private types.
- Use private types in preference to nonprivate types.
- Explicitly export needed operations rather than easing restrictions.

data structures

- Use records to group heterogeneous but related data.
- Consider records to map to I/O device data.
- Record structures should not always be flat. Factor out common parts.
- For a large record structure, group related components into smaller subrecords.
- For nested records, pick element names that read well when inner elements are referenced.
- Differentiate between static and dynamic data. Use dynamically allocated objects with caution.
- Use dynamically allocated data structures only when it is necessary to create and destroy them dynamically or to be able to reference them by different names.
- Do not drop pointers to undeallocated objects.
- Do not leave dangling references to deallocated objects.
- Initialize all access variables and components.
- Do not rely on memory deallocation.
- Deallocate explicitly.
- Use length clauses to specify total allocation size.
- Provide handlers for `STORAGE_ERROR`.

expressions

- Use `'FIRST` or `'LAST` instead of numeric literals to represent the first or last values of a range.
- Use the type or subtype name of the range instead of `'FIRST .. 'LAST`.
- Use array attributes `'FIRST`, `'LAST`, or `'LENGTH` instead of numeric literals for accessing arrays.
- Use the `'RANGE` of the array instead of the name of the index type to express a range.
- Use `'RANGE` instead of `'FIRST .. 'LAST` to express a range.
- Use parentheses to specify the order of subexpression evaluation where operators from different precedence levels are involved, and to clarify expressions.
- Avoid names and constructs that rely on the use of negatives.
- Choose names of flags so they represent states that can be used in positive form.
- Use short-circuit forms of the logical operators.
- Use type qualified expressions instead of type conversions wherever possible.
- Use `<=` and `>=` in relational expressions with real operands instead of `=`.

statements

- Minimize the depth of nested expressions.
- Minimize the depth of nested control structures.
- Try simplification heuristics
- Use slices rather than a loop to copy part of an array.
- Never use an `others` choice in a case statement.
- Do not use ranges of enumeration literals in case statements.
- If you use an if statement instead of a case statement, use marker comments indicating the cases.
- Use for loops whenever possible (when the number of iterations is computable).
- Use while loops when the number of iterations is not computable, but a simple continuation condition can be applied at the top of the loop.
- Use plain loops with exit statements for more complex situations.
- Avoid exit statements in while and for loops.

- Minimize the number of ways to exit a loop.
- Use exit statements to enhance the readability of loop termination code.
- Use `exit when ...` rather than `if ... then exit` whenever possible.
- Review exit statement placement.
- Understand and consider specifying bounds on loops.
- Understand and consider specifying bounds on recursion.
- Do not use goto statements unless you are sure there is no alternative.
- If you must use a goto statement, highlight both it and the label.
- Minimize the number of returns from a subprogram.
- Highlight returns with comments or white space to keep them from being lost in other code.
- Use blocks liberally and for their intended purposes.
- Use an aggregate instead of a sequence of assignments to assign values to all fields of a record.
- Use an aggregate instead of a temporary variable when building a record to pass as an actual parameter.
- Use named component association in aggregates.

visibility

- Minimize using the use clause.
- Localize the effect of the use clauses you must employ.
- Use the renames clause judiciously and purposefully.
- Rename a long fully qualified name to reduce the complexity if it becomes unwieldy (Guideline 3.1.4).
- Rename declarations for visibility purposes rather than using the use clause especially infix operators (Guideline 5.7.1).
- Rename parts when interfacing to reusable components originally written with nondescriptive or inapplicable nomenclature.
- Use a project-wide standard list of abbreviations to rename common packages.
- Limit overloading to widely used subprograms that perform similar actions on arguments of different types.
- Preserve the conventional meaning of overloaded operators (Nissen and Wallis 1984).
- Use "+" to identify adding, joining, increasing, and enhancing kinds of functions.
- Use "-" to identify subtraction, separation, decreasing, and depleting kinds of functions.

using exceptions

- Avoid causing exceptions to be raised when it is easy and efficient to do so.
- Provide handlers for exceptions which cannot be avoided.
- Use exception handlers to enhance readability by separating fault handling from normal execution.
- Do not use exceptions and exception handlers as goto statements.
- Use caution when programming handlers for `others`.
- Provide a handler for `others` in suitable frames to protect against unexpected exceptions being propagated without bound, especially in safety critical systems.
- Use `others` only to catch exceptions you cannot enumerate explicitly, preferably only to flag a potential abort.
- Avoid using `others` during development.
- Handle all exceptions, both user and predefined.

- For every exception that might be raised, provide a handler in suitable frames to protect against undesired propagation outside the abstraction.
- Do not rely on being able to identify the fault raising predefined or implementation-defined exceptions.
- Use blocks to associate localized sections of code with their own exception handlers.

erroneous execution

- Use UNCHECKED_CONVERSION only with the utmost care.
- Isolate the use of UNCHECKED_CONVERSION in package bodies.
- Use UNCHECKED_DEALLOCATION purposefully and carefully.
- Isolate the use of UNCHECKED_DEALLOCATION in package bodies.
- Do not write code whose correct execution depends on the particular parameter passing mechanism used by an implementation.
- Use address clauses to map variables and entries to the hardware device or memory, not to model the FORTRAN "equivalence" feature.
- Do not suppress exception checks during development.
- Minimize suppression of exception checks during operation.
- Initialize all objects prior to use.
- Ensure elaboration of an entity before using it.
- Use function calls in declarations cautiously.

CHAPTER 6
Concurrency

Concurrency exists as either apparent concurrency or real concurrency. In a single processor environment apparent concurrency is the result of interleaved execution of concurrent activities. In a multi-processor environment real concurrency is the result of overlapped execution of concurrent activities.

Concurrent programming is more difficult and error prone than sequential programming. The concurrent programming features of Ada are designed to make it easier to write and maintain concurrent programs which behave consistently and predictably, and avoid such problems as deadlock and starvation. The language features themselves cannot guarantee that programs have these desirable properties. They must be used with discipline and care, a process supported by the guidelines in this chapter.

The correct usage of Ada concurrency features results in reliable, reusable, and portable software. For example, using tasks to model concurrent activities and using the rendezvous for the required synchronization between cooperating concurrent tasks. Misuse of language features results in software that is unverifiable and difficult to reuse or port. For example, using task priorities or delays to manage this synchronization.

Avoid assuming that the rules of good sequential program design can be applied, by analogy, to concurrent programs. For example, while multiple returns from subprograms should be discouraged (Guideline 5.6.8), multiple task exits or termination points are often necessary and desirable.

6.1 TASKING

Many problems map naturally to a concurrent programming solution. By understanding and correctly using the Ada language tasking features you can produce solutions that are independent of target implementation. Tasks provide a means, within the Ada language, of expressing concurrent asynchronous threads of control and relieving programmers from the problem of explicitly controlling multiple concurrent activities.

Tasks cooperate to perform the required activities of the software. Synchronization is required between individual tasks. The Ada rendezvous provides a powerful mechanism for this synchronization.

6.1.1 Tasks

guideline

- Use tasks to model asynchronous entities within the problem domain.
- Use tasks to control or synchronize access to tasks or devices.
- Use tasks to define concurrent algorithms.
- Use tasks to perform cyclic or prioritized activities (NASA 1987).

example

Asynchronous entities are the naturally concurrent objects within the problem domain. These tend to be objects in the problem space that have state, such as elevators in an elevator control system or satellites in a global positioning system. The following is an example for an elevator control system:

```
package ELEVATOR_OBJECTS is
   ...
   type ELEVATOR_STATES is (MOVING, IDLE, STOPPED, AT_FLOOR);
   type UP_DOWN         is (UP, DOWN);
   task type ELEVATORS  is
      entry INITIALIZE;
      entry CLOSE_DOOR;
      entry OPEN_DOOR;
      entry STOP;
      entry IDLE;
      entry START         (DIRECTION       : in     UP_DOWN)
      entry CURRENT_STATE (MY_STATE        :    out ELEVATOR_STATES;
                           CURRENT_LOCATION:    out FLOAT);
   end ELEVATORS;
   ...
end ELEVATOR_OBJECTS;
```

A task DISPLAY_MANAGER that manages updates from multiple concurrent user tasks to a graphic display is an example of a control and synchronization task.

Multiple tasks that implement the decomposition of a large matrix multiplication algorithm is an example of an opportunity for real concurrency in a multi-processor target environment. In a single processor target environment this approach may not be justified.

A task DISPLAY_UPDATE that updates a RADAR display every 30 milliseconds is an example of a cyclic activity supported by a task.

A task PRIORITY_SHUTDOWN that detects an over-temperature condition in a nuclear reactor and performs an emergency shutdown of the systems is an example of a task to support a high priority activity.

rationale

These are the intended uses of tasks. They all revolve around the fact that a task has its own thread of control separate from the main subprogram. The conceptual model for a task is that it is a program with its own virtual processor. This provides the opportunity to model entities from the problem domain in terms more closely resembling those entities, and the opportunity to deal with physical devices on their own terms as a separate concern from the main algorithm of the application. Tasks also allow the programming of naturally concurrent activities in their own terms, and they can be mapped to multiple processors when these are available.

Resources shared between multiple tasks, such as devices and abstract data structures, require control and synchronization since their operations are not atomic. In our display manager example, drawing a circle on the display may require that many low level operations be performed without interruption by another task. The display manager would ensure that no other task accesses the display until all these operations are complete.

Cyclic and prioritized tasks allow the programmer to ensure that these critical activities occur when required without the complexity of explicit scheduling them within the application.

6.1.2 Anonymous Task Types

guideline

- Use anonymous task types for single instances.

example

The example below illustrates the syntactic differences between the kinds of tasks discussed here. BUFFER is static and has a name, but its type is anonymous. Because it is declared explicitly, the task type BUFFER_MANAGER is not anonymous. CHANNEL is static and has a name, and its type is not anonymous. Like all dynamic objects, ENCRYPTED_PACKET_QUEUE.ALL is essentially anonymous, but its type is not.

```
task BUFFER                is ...
task type BUFFER_MANAGER   is ...
type REPLACEABLE_BUFFER    is access BUFFER_MANAGER;
...
ENCRYPTED_PACKET_QUEUE : REPLACEABLE_BUFFER;
CHANNEL                : BUFFER_MANAGER;
...
ENCRYPTED_PACKET_QUEUE := new BUFFER_MANAGER;
```

rationale

The use of named tasks of anonymous type avoids a proliferation of task types that are only used once, and the practice communicates to maintainers that there are no other task objects of that type. If the need arises later to have additional tasks of the same type, then the work required to convert a named task to a task type is minimal. It involves including task type declarations and deciding whether static or dynamic tasks should be used.

The consistent and logical use of task types, when necessary, contributes to understandability. Identical tasks can be derived from a common task type. Dynamically allocated task structures are necessary when you must create and destroy tasks dynamically or when you must reference them by different names.

6.1.3 Dynamic Tasks

guideline

- Use caution with dynamically allocated task objects.
- Avoid referencing terminated tasks through their aliases.
- Avoid disassociating a task from all names.

example

The approach used in the example below is not recommended. The example shows why caution is required with dynamically allocated task objects. It illustrates an attempt to call an entry in an aborted task after the abort operation was applied to the alias. In the example, the limited number of trackable radar targets are tasks continuously updating their positions based on previous position and velocity until corrected by a new scan. Out-of-range targets are dropped (through use of the abort statement).

Execute these lines in subprograms in the radar package first:

```
TARGET (LATEST_ACQUISITION) := new RADAR_TRACK;
TARGET (LATEST_ACQUISITION).INITIALIZE
      (SELF     =>      TARGET (LATEST_ACQUISITION),
       VELOCITY => ...                              ,
       POSITION => ...                              );
```

Execute these lines in the body of task type RADAR_TRACK next. They are not inside an accept statement:

```
NEW_POSITION := INTEGRATE (POSITION, VELOCITY);
if OUT_OF_RANGE (NEW_POSITION) then
   abort SELF; --notice abort
end if;
```

Execute this line in a subprogram in the radar package last. This line can raise TASKING_ERROR due to calling an entry of an aborted task:

```
TARGET (SCAN_HIT).CORRECT_READINGS (POSITION, VELOCITY);
```

rationale

A dynamically allocated task object is a task object created by the execution of an allocator. They can be used to avoid limiting the number of allocatable objects. This is useful when the upper limit is unknown or for performance purposes.

Allocated task objects referenced by access variables allow you to generate *aliases;* multiple references to the same object. Anomalous behavior can arise when you reference an aborted task by another name.

A dynamically allocated task that is not associated with a name (a "dropped pointer") cannot be referenced for the purpose of making entry calls, nor can it be the direct target of an abort statement (see Guideline 5.4.3).

6.1.4 Priorities

guideline

- Do not rely on pragma PRIORITY to perform precise scheduling.

example

For example, let the tasks have the following priorities:

```
task T1 ... pragma PRIORITY (HIGH)   ... SERVER.OPERATION ...
task T2 ... pragma PRIORITY (MEDIUM) ... SERVER.OPERATION ...
task SERVER                          ... accept OPERATION ...
```

At some point in its execution, T1 is blocked. Otherwise, we would not expect T2 or SERVER to ever get anything done. If T1 is blocked, it is possible for T2 to reach its call to SERVER's entry (OPERATION) before T1. Suppose this has happened and that T1 now makes its entry call before SERVER has a chance to accept T2's call.

This is the timeline of events so far:

T1 blocks
T2 calls SERVER.OPERATION
T1 unblocks
T1 calls SERVER.OPERATION
DOES SERVER accept the call from T1 or from T2?

Some people might expect that, due to its higher priority, T1's call would be accepted by SERVER before that of T2. However, entry calls are queued *in first-in-first-out* (FIFO) order and not queued in order of priority. Therefore, the synchronization between T1 and SERVER is not affected by T1's priority. As a result, the call from T2 is accepted first. This is a form of *priority inversion*.

A solution to this might be to provide an entry for a HIGH priority user and an entry for a MEDIUM priority user.

```
task SERVER is
   entry OPERATION_HIGH_PRIORITY;
   entry OPERATION_MEDIUM_PRIORITY;
   ...
end SERVER;
task body SERVER is
   procedure OPERATION; -- provides functions from previous example
   ...
begin
   loop
      select
         accept OPERATION_HIGH_PRIORITY do
            OPERATION;
         end OPERATION_HIGH_PRIORITY;
      else
         select
            accept OPERATION_HIGH_PRIORITY do
                  OPERATION;
            end OPERATION_HIGH_PRIORITY;
         or
            accept OPERATION_MEDIUM_PRIORITY do
                  OPERATION;
            end OPERATION_MEDIUM_PRIORITY;
         or
            terminate;
         end select;
      end select;
   end loop;
   ...
end SERVER;
```

However, in this approach T1 still waits for one execution of OPERATION when T2 has already gained control of the task SERVER. In addition, the approach increases the communication complexity (see Guideline 6.2.6).

rationale

The pragma PRIORITY allows relative priorities to be placed on tasks to accomplish scheduling. Precision becomes a critical issue with hard-deadline scheduling. However, there are certain problems associated with using priorities that warrant caution.

Priority inversion occurs where lower priority tasks are given service while higher priority tasks remain blocked. In the example above, this occured because entry queues are serviced in FIFO order, not by priority. There is another situation referred to as a *race condition*. A program like the one in the first example might often behave as expected as long as T1 calls SERVER.OPERATION only when T2 is not already using SERVER.OPERATION or waiting. You cannot rely on T1 always winning the race, since that behavior would be due more to fate than to the programmed priorities. Race conditions change when either adding code to some unrelated task or porting this code to new target. Task priorities are not a means of achieving mutual exclusion.

There is work being done to address such problems, including Rate Monotonic Analysis (Sha, L. and J. B. Goodenough, 1989).

note

Priorities are used to control when tasks run relative to one another, i.e. when both tasks are eligible to execute, that is, not blocked waiting at an entry, the highest priority task will be given precedence. However, the most critical tasks in an application do not always have the highest priority. For example, support tasks or tasks with small periods may have higher priorities, because they need to run frequently. Any blocking might cause a bottleneck.

6.1.5 Delay Statements

guideline

- Do not depend on a particular delay being achievable (Nissen and Wallis 1984).
- Do not use a busy waiting loop instead of a delay.
- Design to limit polling to those cases where absolutely necessary.
- Do not use knowledge of the execution pattern of tasks to achieve timing requirements.

example

In the following example the period drifts over time:

```
PERIODIC:
   loop
      delay INTERVAL;
      ... -- some actions
   end loop PERIODIC;
```

The following example shows how to compensate for the incorrectness of the delay statement. This approach works well when the periodic requirement can be satisfied with an average period. Periodic tasks based on a delay can drift from their schedule. Prevention of this drift can be achieved by calculating the next time-to-occur based on the actual time of the current execution. The following example illustrates this tactic.

```
NO_DRIFT:
   declare
      use CALENDAR;
      -- INTERVAL is a global constant of type DURATION
      NEXT_TIME : TIME := CLOCK + INTERVAL;
   begin
      PERIODIC:
         loop
            delay NEXT_TIME - CLOCK;
            ...  -- some actions
            NEXT_TIME := NEXT_TIME + INTERVAL;
         end loop PERIODIC;
   end NO_DRIFT;
```

rationale

The Ada language definition only guarantees that the delay time is a minimum. The meaning of a `delay` statement is that the task is not scheduled for execution before the interval has expired. In other words, a task becomes eligible to resume execution as soon as the amount of time has passed. However, there is no guarantee of when (or if) it is scheduled after that time. This must be the case in light of the potentially ever changing task and priority mix with which the scheduling algorithm must deal.

A busy wait can interfere with processing by other tasks. It can consume the very processor resource necessary for completion of the activity for which it is waiting. Even a loop with a delay can have the impact of busy waiting if the planned wait is significantly longer then the delay interval. If a task has nothing to do, it should be blocked at an accept or select statement.

Using knowledge of the execution pattern of tasks to achieve timing requirements is nonportable since the underlying scheduling algorithm may change.

6.2 COMMUNICATION

The need for tasks to communicate gives rise to most of the problems that make concurrent programming so difficult. Used properly, Ada's intertask communication features can improve the reliability of concurrent programs; used thoughtlessly they can introduce subtle errors that can be difficult to detect and correct.

6.2.1 Efficient Task Communications

guideline

- Minimize the work performed during a rendezvous.
- Minimize the work performed in the selective wait loop of a task.

example

In the following example the statements in the accept block are performed as part of the execution of both the caller task and the task SERVER which contains OPERATION. The statements after the accept block are executed before SERVER can go back to accept additional calls to OPERATION.

```
loop
   select
      accept OPERATION (...) do
         -- These statements are executed during rendezvous.
         -- Both caller and SERVER are blocked during this time.
         ...
      end OPERATION;
      ...
      -- These statements are not executed during rendezvous.
      -- Their execution delays getting back to the accept.
   end select;
   -- These statements are also not executed during rendezvous.
   -- Their execution delays getting back to the accept.
end loop;
```

rationale

Only work that needs to be performed during a rendezvous, such as saving or generating parameters, should be allowed in the statements following the accept statement to minimize the time required to rendezvous.

In the example, callers to OPERATION are blocked while SERVER is executing statements before returning to the accept. This work should be limited to the services provided by SERVER.

note

In some cases, additional functions other than the services provided may be added to a task. For example, a task controlling a communication device may be responsible for a periodic BUILT_IN_TEST function to ensure that the device is operating correctly. This type of addition should be done with care realizing that the response time of the task is impacted.

6.2.2 Defensive Task Communication

guideline

- Provide a handler for exception PROGRAM_ERROR wherever there is no `else` in a selective wait statement (Honeywell 1986).
- Make systematic use of handlers for TASKING_ERROR.
- Be prepared to handle exceptions during a rendezvous.

example

This block allows recovery from exceptions raised while attempting to communicate a command to a task controlling the throttle.

```
ACCELERATE:
   begin
      THROTTLE.INCREASE (STEP);
   exception
      when TASKING_ERROR =>
         ...
      when CONSTRAINT_ERROR
         | NUMERIC_ERROR =>
         ...
      when THROTTLE_TOO_WIDE =>
         ...
      ...
   end ACCELERATE;
```

In this select statement, if all the guards happen to be closed, the program can continue by executing the else part. There is no need for a handler for PROGRAM_ERROR. Other exceptions can still be raised while evaluating the guards or attempting to communicate.

```
BUFFER:
   begin
      select
         when ... =>
            accept ...
      or
         when ... =>
            accept ...
      else
         ...
      end select;
   exception
      when CONSTRAINT_ERROR
         | NUMERIC_ERROR =>
         ...
      ...
   end BUFFER;
```

In this select statement, if all the guards happen to be closed, exception PROGRAM_ERROR will be raised. Other exceptions can still be raised while evaluating the guards or attempting to communicate.

```
BUFFER:
   begin
      select
         when ... =>
            accept ...
      or
         when ... =>
            delay ...
      end select;
   exception
      when PROGRAM_ERROR =>
         ...
      when CONSTRAINT_ERROR
         | NUMERIC_ERROR =>
         ...
      ...
   end BUFFER;
```

rationale

The exception PROGRAM_ERROR is raised if a selective wait statement (select statement containing accepts) is reached, all of whose alternatives are closed (i.e., the guards evaluate to FALSE and there are no alternatives without guards), unless there is an `else part`. When all alternatives are closed, the task can never again progress, so there is by definition an error in its programming. You must be prepared to handle this error should it occur.

Since an else part cannot have a guard, it can never be closed off as an alternative action, thus its presence prevents PROGRAM_ERROR. However, an `else part`, a delay alternative, and a terminate alternative are all mutually exclusive, so you will not always be able to provide an else part. In these cases, you must be prepared to handle PROGRAM_ERROR.

The exception TASKING_ERROR can be raised in the calling task whenever it attempts to communicate. There are many situations permitting this. Few of them are preventable by the calling task.

If an exception is raised during a rendezvous and not handled in the accept statement, it is propagated to both tasks and must be handled in two places. See Section 5.8.

note

There are other ways to prevent PROGRAM_ERROR at a selective wait. These involve leaving at least one alternative unguarded, or proving that at least one guard will evaluate TRUE under all circumstances. The point here is that you, or your successors, will make mistakes in trying to do this, so you should prepare to handle the inevitable exception.

6.2.3 Attributes 'COUNT, 'CALLABLE and 'TERMINATED

guideline

- Do not depend on the values of the task attributes 'CALLABLE or 'TERMINATED (Nissen and Wallis 1984).
- Do not depend on attributes to avoid TASKING_ERROR on an entry call.
- Do not depend on the value of the entry attribute 'COUNT.

example

In the following examples INTERCEPT'CALLABLE is a boolean indicating if a call can be made to the task INTERCEPT without raising the exception TASKING_ERROR. LAUNCH'COUNT indicates the number of callers currently waiting at entry LAUNCH. INTERCEPT'TERMINATED is a boolean indicating if the task INTERCEPT is in terminated state.

This task is badly programmed because it relies upon the values of the 'COUNT attributes not changing between evaluating and acting upon them.

```
-------------------------------------------------------------------------
task body INTERCEPT is
   ...
   select
      when (LAUNCH'COUNT > 0) and
           (RECALL'COUNT = 0)      =>
         accept LAUNCH;
         ...
   or
      accept RECALL;
      ...
   end select;
   ...
-------------------------------------------------------------------------
```

If the following code is preempted between evaluating the condition and initiating the call, the assumption that the task is still callable may no longer be valid.

```
   if INTERCEPT'CALLABLE then
      INTERCEPT.RECALL;
   ...
```

rationale

Attributes 'CALLABLE, TERMINATED, and 'COUNT are all subject to race conditions. Between the time you reference an attribute and the time you take action the value of the attribute may change. Attributes 'CALLABLE and 'TERMINATED convey reliable information once they become FALSE and TRUE, respectively. If 'CALLABLE is FALSE, you can expect the callable state to remain constant. If 'TERMINATED is TRUE, you can expect the task to remain terminated. Otherwise, 'TERMINATED and 'CALLABLE can change between the time your code tests them and the time it responds to the result.

The Ada Language Reference Manual (Department of Defense 1983) itself warns about the asynchronous increase and decrease of the value of 'COUNT. A task can be removed from an entry queue due to execution of an abort statement as well as expiration of a timed entry call. The use of this attribute in guards of a selective wait statement may result in the opening of alternatives which should not be opened under a changed value of 'COUNT.

exceptions

Use extreme care.

6.2.4 Shared Variables

guideline

- Use the rendezvous mechanism, not shared variables, to pass data between tasks.
- Do not use shared variables as a task synchronization device.

example

This code will either print the same line more than once, fail to print some lines, or print garbled lines (part of one line followed by part of another) nondeterministically.

```
-------------------------------------------------------------------------
task body ROBOT_ARM_DRIVER is
   ...
begin
   loop
      CURRENT_COMMAND := COMMAND;
      -- send to device
   end loop;
end ROBOT_ARM_DRIVER;
-------------------------------------------------------------------------
task body STREAM_SERVER is
   ...
begin
   loop
      STREAM_READ (STREAM_FILE, COMMAND);
   end loop;
end STREAM_SERVER;
-------------------------------------------------------------------------
```

rationale

There are many techniques for protecting and synchronizing data access. You must program most of them yourself to use them. It is difficult to write a program that shares data correctly. If it is not done correctly, the reliability of the program suffers. Ada provides the rendezvous to support synchronization and communication of information between tasks. Data that you might be tempted to share can be put into a task body with read and write entries to access it.

The example above has a race condition requiring perfect interleaving of execution. This code can be made more reliable by introducing a flag that is set by SPOOL_SERVER and reset by LINE_PRINTER_DRIVER. An `if (condition flag) then delay ... else` would be added to each task loop in order to ensure that the interleaving is satisfied. However, notice that this approach requires a delay and the associated rescheduling. Presumably this rescheduling overhead is what is being avoided by not using the rendezvous.

exceptions

For some required synchronizations the rendezvous may not meet time constraints. Each case should be analyzed in detail to justify the use of pragma SHARED, which presumably has less overhead than the rendezvous. Be careful to correctly implement a data access synchronization technique. Without great effort you might get it wrong. Pragma SHARED can serve as an expedient against poor run time support systems. Do not always use this as an excuse to avoid the rendezvous because implementations are allowed to ignore pragma SHARED (Nissen and Wallis 1984). Pragma SHARED affects only those objects which storage and retrieval are implemented as indivisible operations. Also, pragma SHARED can only be used for variables of scalar or access type.

note

As pointed out above, a guarantee of noninterference may be difficult with implementations that ignore pragma SHARED. If you must share data, share the absolute minimum amount necessary, and be especially careful. As always, encapsulate the synchronization portions of code.

The problem is with variables. Constants, such as tables fixed at compile time, may be safely shared between tasks.

For further reading on shared variables, see (Dewar, R., 1990).

6.2.5 Tentative Rendezvous Constructs

guideline

- Use caution with conditional entry calls.
- Use caution with selective waits with else parts.
- Do not depend upon a particular delay in timed entry calls.
- Do not depend upon a particular delay in selective waits with delay alternatives.

example

The conditional entry call in the following code results in a race condition that may degenerate into a busy waiting loop. The task CURRENT_POSITION containing entry REQUEST_NEW_COORDINATES may never execute if this task has a higher priority than CURRENT_POSITION, because this task doesn't release the processing resource.

```
...
loop
   select
      CURRENT_POSITION.REQUEST_NEW_COORDINATES (...);
      -- calculate target location based on new coordinates
      ...
   else
      -- calculate target location based on last locations
      ...
   end select
end loop
```

The addition of a delay as shown may allow CURRENT_POSITION to execute until it reaches an accept for REQUEST_NEW_COORDINATES.

```
...
loop
   select
      CURRENT_POSITION.REQUEST_NEW_COORDINATES (...);
      -- calculate target location based on new coordinates
      ...
   else
      -- calculate target location based on last locations
      ...
      delay NEXT_EXECUTE-CLOCK;
      NEXT_EXECUTE := NEXT_EXECUTE + PERIOD;
   end select
end loop
```

The following selective wait with else again does not degenerate into a busy wait loop only because of the addition of a delay statement.

```
loop
   delay NEXT_EXECUTE - CLOCK;
   select
      accept GET_NEW_MESSAGE (...)
         do
            -- copy message to parameters
            ...
         end GET_NEW_MESSAGE;
   else
      -- perform BUILD_IN_TEST Functions
      ...
   end select;
   NEXT_EXECUTE := NEXT_EXECUTE + TASK_PERIOD;
end loop;
```

The following timed entry call may be considered an unacceptable implementation if lost communications with the reactor for over 25 milliseconds results in a critical situation.

```
loop
   select
      GET_REACTOR_STATUS;
   or
      delay 0.025;
      -- lost communication for more that 25 milliseconds
      EMERGENCY_SHUTDOWN;
   end select;
   -- process reactor status
   ...
   delay NEXT_TIME - CLOCK;
   NEXT_TIME := NEXT_TIME + PERIOD;
end loop;
```

In the following selective wait with delay example the accuracy of the coordination calculation function may be bounded by time. For example, the required accuracy cannot be obtained unless PERIOD is within + or − 0.005 seconds.

```
loop
   select
      accept REQUEST_NEW_COORDINATES (...) do
         -- copy coordinates to parameters
      end REQUEST_NEW_COORDINATES;
   or
      delay NEXT_EXECUTE;
   end select;
   NEXT_EXECUTE := NEXT_EXECUTE + PERIOD;
   -- Read Sensors
   -- execute coordinate transformations
end loop;
```

rationale

Using of these constructs always poses a risk of race conditions. Using them in loops, particularly with poorly chosen task priorities, can have the effect of busy waiting.

These constructs are very much implementation dependent. For conditional entry calls and selective waits with else parts, the Ada Language Reference Manual (Department of Defense 1983) does not define "immediately." For timed entry calls and selective waits with delay alternatives, implementors may have ideas of time that differ from each other and from your own. Like the delay statement, the delay alternative on the select construct might wait longer than the time required (see Guideline 6.1.5).

6.2.6 Communication Complexity

guideline

- Minimize the number of accept and select statements per task.
- Minimize the number of accept statements per entry.
- Minimize the number of statements within an accept.

example

```
-- use
       accept A;
       if MODE_1 then
          -- do one thing
       else -- MODE_2
          -- do something different
       end if;

-- rather than
       if MODE_1 then
          accept A do
             -- do one thing
                ...
       else -- MODE_2
          accept A do
             -- do something different
       end if;
```

rationale

This guideline is motivated by reduction of conceptual complexity. With small numbers of accept or select statements, the programmers of the task and calling units need not reason about the circumstances of an entry call executing different code sequences dependent on the task's local state.

A large number of accept and select statements carries with it a large amount of intertask communication, with its inevitable overhead. It could be that tasks which need to communicate very frequently are poorly designed. The communication overhead should, in general, be insignificant compared with the independent, parallel computation.

The calling task is blocked for the duration of the rendezvous. During the rendezvous, the calling task should have to wait only if data is to be copied or returned. If additional work needs to be done as a result of the accept construct, place it after the rendezvous.

6.3 TERMINATION

The ability of tasks to interact with each other using Ada's intertask communication features makes it especially important to manage planned or unplanned (e.g., in response to a catastrophic exception condition) termination in a disciplined way. To do otherwise can lead to a proliferation of undesired and unpredictable side effects as a result of the termination of a single task.

6.3.1 Avoiding Termination

guideline

- Place an exception handler for rendezvous within main tasking loop.

example

In the following example an exception raised using the primary sensor is used to change MODE TO DEGRADED still allowing execution of the system.

```
...
loop
   if MODE = PRIMARY then
      select
         CURRENT_POSITION_PRIMARY.REQUEST_NEW_COORDINATES (...);
      end select
   else
      select
         CURRENT_POSITION_BACKUP.REQUEST_NEW_COORDINATES (...);
      end select
   end if;
   exception
      when PROGRAM_ERROR =>
         MODE := DEGRADED;
end loop
```

rationale

Allowing a task to terminate may not support the requirements of the system. Without an exception handler for the rendezvous within the main task loop, the functions of the task may not be performed.

6.3.2 Normal Termination

guideline

- Do not create non-terminating tasks unintentionally.
- Explicitly shut down tasks dependent on library packages.
- Use a select statement rather than an accept statement alone.
- Provide a terminate alternative for every selective wait that does not require an else part or a delay.

example

This task will never terminate:

```
-------------------------------------------------------------------------
task body MESSAGE_BUFFER is
   ...
begin -- MESSAGE_BUFFER
   loop
      select                  -- Circular buffer not empty
         when (HEAD /= TAIL) =>
            accept RETRIEVE ( ... );
      or                      -- Circular buffer not full
         when not (((HEAD = LOWER_BOUND              ) and then
                    (TAIL = UPPER_BOUND              )         ) or else
                   ((HEAD/= LOWER_BOUND              ) and then
                    (TAIL = BUFFER'RANGE'PRED (HEAD))         )       ) =>
            accept STORE ( ... );
      end select;
   end loop;
end MESSAGE_BUFFER;
-------------------------------------------------------------------------
```

rationale

A nonterminating task is a task whose body consists of a nonterminating loop with no selective wait with terminate, or a task that is dependent on a library package. Execution of a subprogram or block containing a task cannot complete until the task terminates. Any task that calls a subprogram containing a nonterminating task will be delayed indefinitely.

The effect of unterminated tasks at the end of program execution is undefined. A task dependent on a library package cannot be forced to terminate using a selective wait construct with terminate alternative and should be terminated explicitly during program shutdown. One way to terminate tasks dependent on library packages is to provide them with exit entries. Have the main subprogram call the exit entry just before it terminates.

Execution of an accept statement or of a selective wait statement without an else part, a delay, or a terminate alternative cannot proceed if no task ever calls the entry(s) associated with that statement. This could result in deadlock. Following this guideline entails programming multiple termination points

in the task body. A reader can easily "know where to look" for the normal termination points in a task body. The termination points are the end of the body's sequence of statements, and alternatives of select statements.

exceptions

If you are simulating a cyclic executive, you may need a scheduling task that does not terminate. It has been said that no real-time system should be programmed to terminate. This is extreme. Systematic shutdown of many real-time systems is a desirable safety feature.

If you are considering programming a task not to terminate, be certain that it is not a dependent of a block or subprogram from which the task's caller(s) will ever expect to return. Since entire programs can be candidates for reuse (see Chapter 8), note that the task (and whatever it depends upon) will not terminate. Also be certain that for any other task that you do wish to terminate, its termination does not await this task's termination. Reread and fully understand the (Ada Reference Manual 1983, § 9.4) on "Task Dependence - Termination of Tasks."

6.3.3 The Abort Statement

guideline

- Avoid using the abort statement.

example

If required in the application, provide a task entry for orderly shutdown.

rationale

When an abort statement is executed, there is no way to know what the targeted task was doing beforehand. Data for which the target task is responsible may be left in an inconsistent state. The overall effect on the system of aborting a task in such an uncontrolled way requires careful analysis.

Tasks are not aborted until they reach a synchronization point such as beginning or end of elaboration, delay, and accept statement or an entry call, selective wait, task allocation, or execution of an exception handler. Consequently, the abort statement may not release processor resources as soon as you may expect. It also may not stop a runaway task because the task may be executing an infinite loop containing no synchronization points.

6.3.4 Abnormal Termination

guideline

- Place an exception handler for `others` at the end of a task body.
- Have each exception handler at the end of a task body report the task's demise.

example

This is one of many tasks updating the positions of blips on a radar screen. When started, it is given part of the name by which its parent knows it. Should it terminate due to an exception, it signals the fact in one of its parent's data structures.

```
-------------------------------------------------------------------------
task body TRACK is
   MY_INDEX : TRACKS := NEUTRAL;
   ...
begin -- TRACK
   select
      accept START (WHO_AM_I : TRACKS) do
         MY_INDEX := WHO_AM_I;
      end START;
   or
      TERMINATE;
   end select;
   ...
exception
   when others =>
      if MY_INDEX /= NEUTRAL then
         STATION (MY_INDEX).STATUS := DEAD;
      end if;
end TRACK;
-------------------------------------------------------------------------
```

rationale

A task will terminate if an exception is raised within it for which it has no handler. In such a case, the exception is not propagated outside of the task (unless it occurs during a rendezvous). The task simply dies with no notification to other tasks in the program. Therefore, providing exception handlers within the task, and especially a handler for `others`, ensures that a task can regain control after an exception occurs. If the task cannot proceed normally after handling an exception, this at least affords it the opportunity to notify other tasks of its demise and to shut itself down cleanly.

6.4 SUMMARY

tasking

- Use tasks to model asynchronous entities within the problem domain.
- Use tasks to control or synchronize access to tasks or devices.
- Use tasks to define concurrent algorithms.
- Use tasks to perform cyclic or prioritized activities.
- Use anonymous task types for single instances.
- Use caution with dynamically allocated task objects.
- Avoid referencing terminated tasks through their aliases.
- Avoid disassociating a task from all names.
- Do not rely on pragma `PRIORITY` to perform precise scheduling.
- Do not depend on a particular delay being achievable.
- Do not use a busy waiting loop instead of a delay.
- Design to limit polling to those cases where absolutely necessary.
- Do not use knowledge of the execution pattern of tasks to achieve timing requirements.

communication

- Minimize the work performed during a rendezvous.
- Minimize the work performed in the selective wait loop of a task.
- Provide a handler for exception `PROGRAM_ERROR` wherever there is no `else` in a selective wait statement.
- Make systematic use of handlers for `TASKING_ERROR`.
- Be prepared to handle exceptions during a rendezvous.
- Do not depend on the values of the task attributes `'CALLABLE` or `'TERMINATED`.

- Do not depend on attributes to avoid TASKING_ERROR on an entry call.
- Do not depend on the value of the entry attribute 'COUNT.
- Use the rendezvous mechanism, not shared variables, to pass data between tasks.
- Do not use shared variables as a task synchronization device.
- Use caution with conditional entry calls.
- Use caution with selective waits with else parts.
- Do not depend upon a particular delay in timed entry calls.
- Do not depend upon a particular delay in selective waits with delay alternatives.
- Minimize the number of accept and select statements per task.
- Minimize the number of accept statements per entry.
- Minimize the number of statements within an accept.

termination

- Place an exception handler for rendezvous within main tasking loop.
- Do not create non-terminating tasks unintentionally.
- Explicitly shut down tasks dependent on library packages.
- Use a select statement rather than an accept statement alone.
- Provide a terminate alternative for every selective wait that does not require an else part or a delay.
- Avoid using the abort statement.
- Place an exception handler for others at the end of a task body.
- Have each exception handler at the end of a task body report the task's demise.

CHAPTER 7
Portability

The manner in which the Ada language has been defined and tightly controlled is intended to provide considerable aid in the portability of Ada programs. In most programming languages, different dialects are prevalent as vendors extend or dilute a language for various reasons such as conformance to a programming environment or to a particular application domain. The Ada Compiler Validation Capability (ACVC) was developed by the U.S. Department of Defense at the Ada Validation Facility, ASD/SIDL, Wright-Patterson Air Force Base to ensure that implementors strictly adhered to the Ada standard. Although the ACVC mechanism is very beneficial and does eliminate many portability problems that plague other languages, there is a tendency for new Ada users to expect it to eliminate all portability problems; it definitely does not. Certain areas of Ada are not covered by validation. The semantics of Ada leave certain details to the implementor. The implementor's choices with respect to these details affect portability.

There are some general principles to enhancing portability exemplified by many of the guidelines in this chapter. They are:

- Recognize those Ada constructs that may adversely impact portability.
- Avoid the use of these constructs where possible.
- Localize and encapsulate nonportable features of a program if their use is essential.
- Highlight the use of constructs that may cause portability problems.

These guidelines cannot be applied thoughtlessly. Many of them involve a detailed understanding of the Ada model and its implementation. In many cases you will have to make carefully considered tradeoffs between efficiency and portability. Reading this chapter should improve your insight into the issues involved.

The material in this chapter was largely acquired from three sources: the Ada Run Time Environment Working Group (ARTEWG) Catalogue of Ada Run Time Implementation Dependencies (ARTEWG 1986); the Nissen and Wallis book on Portability and Style in Ada (Nissen and Wallis 1984); and a paper written for the U.S. Air Force by SofTech on Ada Portability Guidelines (Pappas 1985). The last of these sources (Pappas 1985) encompasses the other two and provides an in-depth explanation of the issues, numerous examples, and techniques for minimizing portability problems. (Conti 1987) is a valuable reference for understanding the latitude allowed for implementors of Ada and the criteria often used to make decisions.

The purpose of this chapter is to provide a summary of portability issues in the guideline format of this book. The chapter does not include all issues identified in the references, rather the most significant. For an in-depth presentation, (see Pappas 1985). A few additional guidelines are presented here and others are elaborated upon where the authors' experience is applicable. For further reading on Ada I/O portability issues, see (Matthews 1987). None of its suggestions were included herein, but it may be of interest.

The goal of this chapter is to aid you in writing portable Ada code. There are fewer exceptions provided for the guidelines because many of the guidelines are rules of thumb that have been used effectively in the past.

Some of the guidelines in this chapter cross reference and place stricter constrains on other guidelines in this book. These constraints apply when portability is being emphasized.

7.1 FUNDAMENTALS

This section introduces some generally applicable principles of writing portable Ada programs. It includes guidelines about the assumptions you should make with respect to a number of Ada features and their implementations, and guidelines about the use of other Ada features to ensure maximum portability.

7.1.1 Global Assumptions

guideline

- Make considered assumptions about the support provided for the following on potential target platforms:
  - Number of bits available for type INTEGER.
  - Number of decimal digits of precision available for floating point types.
  - Number of bits available for fixed-point types.
  - Number of characters per line of source text.
  - Number of bits for `universal_integer` expressions.
  - Number of seconds for the range of DURATION.
  - Number of milliseconds for DURATION'SMALL.

instantiation

These are minimum values (or minimum precision in the case of DURATION'SMALL) that a project or application might assume that an implementation provides. There is no guarantee that a given implementation provides more than the minimum, so these would be treated by the project or application as maximum values also.

- 16 bits available for type INTEGER.
- 6 decimal digits of precision available for floating point types.
- 32 bits available for fixed-point types.
- 72 characters per line of source text.
- 16 bits for `universal_integer` expressions.
- `-86_400 .. 86_400` seconds (1 day) for the range of DURATION.
- 20 milliseconds for DURATION'SMALL.

rationale

Some assumptions must be made with respect to certain implementation dependent values. The exact values assumed should cover the majority of the target equipment of interest. Choosing the lowest common denominator for values improves portability.

note

Of the microcomputers currently available for incorporation within embedded systems, 16-bit and 32-bit processors are prevalent. Although 4-bit and 8-bit machines are still available, their limited memory addressing capabilities make them unsuited to support Ada programs of any size. Using current representation schemes, 6 decimal digits of floating point accuracy implies a representation mantissa at least 21 bits wide, leaving 11 bits for exponent and sign within a 32-bit representation. This correlates with the data widths of floating point hardware currently available for the embedded systems market. A 32-bit minimum on fixed-point numbers correlates with the accuracy and storage requirements of floating point numbers.

The 72-column limit on source lines in the example is an unfortunate hold-over from the days of Hollerith punch cards with sequence numbers. There may still be machinery and software used in manipulating source code that are bound to assumptions about this maximum line length. The 16-bit example for `universal_integer` expressions matches that for INTEGER storage.

The values for the range and accuracy of values of the predefined type DURATION are the limits expressed in the Ada Language Reference Manual (Department of Defense 1983, § 9.6). You should not expect an implementation to provide a wider range or a finer granularity.

7.1.2 Actual Limits

guideline

- Determine the actual properties and limits of the Ada implementation(s) you are using.

rationale

The Ada model may not match exactly with the underlying hardware, so some compromises may have been made in the implementation. Check to see if they could affect your program. Particular implementations may do "better" than the Ada model requires while some others may be just minimally acceptable. Arithmetic is generally implemented with a precision higher than the storage capacity (this is implied by the Ada type model for floating point). Different implementations may behave differently on the same underlying hardware.

7.1.3 Comments

guideline

- Use highlighting comments for each package, subprogram and task where any nonportable features are present.
- For each nonportable feature employed, describe the expectations for that feature.

example

```
------------------------------------------------------------------------
with SYSTEM;
package MEMORY_MAPPED_IO is
-- WARNING - This package is implementation specific.
-- It uses absolute memory addresses to interface with the I/O system.
-- It assumes a particular printer's line length.
-- Change memory mapping and printer details when porting.

   PRINTER_LINE_LENGTH : constant := 132;
   type DATA is array (1..PRINTER_LINE_LENGTH) of CHARACTER;
   procedure WRITE_LINE (LINE : in DATA);
end MEMORY_MAPPED_IO;
------------------------------------------------------------------------package
body MEMORY_MAPPED_IO is
   ---------------------------------------------------------------------
procedure WRITE_LINE (LINE : in DATA) is
      BUFFER : DATA;
      for BUFFER use at SYSTEM.PHYSICAL_ADDRESS (16#200#);
   begin
      -- perform output operation through specific memory locations.
   end WRITE_LINE;
   ---------------------------------------------------------------------
end MEMORY_MAPPED_IO;
------------------------------------------------------------------------
```

rationale

The explicit commentation of each breach of portability will raise its visibility and aid in the porting process. A description of the non-portable feature's expectations covers the common case where vendor documentation of the original implementation is not available to the person performing the porting process.

7.1.4 Main Subprogram

guideline

- Avoid using any implementation features associated with the main subprogram (e.g., allowing parameters to be passed).

rationale

The Ada Language Reference Manual (Department of Defense 1983) places very few requirements on the main subprogram assuming the simplest case will increase portability. That is, assume you may only use a parameterless procedure as a main program. Some operating systems are capable of acquiring and interpreting returned integer values near zero from a function, but many others cannot. Further, many real-time, embedded systems will not be designed to terminate, so a function or a procedure having parameters with modes `out` or `in out` will be inappropriate to such applications.

This leaves procedures with `in` parameters. Although some operating systems can pass parameters in to a program as it starts, others cannot. Also, an implementation may not be able to perform type checking on such parameters even if the surrounding environment is capable of providing them. Finally, real-time, embedded applications may not have an "operator" initiating the program to supply the parameters, in which case it would be more appropriate for the program to have been compiled with a package containing the appropriate constant values or for the program to read the necessary values from switch settings or a downloaded auxiliary file. In any case, the variation in surrounding initiating environments is far too great to depend upon the kind of last-minute (program) parameterization implied by (subprogram) parameters to the main subprogram.

7.1.5 Encapsulating Implementation Dependencies

guideline

- Encapsulate hardware and implementation dependencies in a package.
- Clearly indicate the objectives if machine or solution efficiency is the reason for hardware or implementation dependent code.
- Develop specific bodies for specific applications to meet particular needs or constraints after porting.
- Isolate interrupt receiving tasks into implementation dependent packages.

example

See Guideline 7.1.3.

rationale

Encapsulating hardware and implementation dependencies in a package allows the remainder of the code to ignore them and thus to be fully portable. It also localizes the dependencies, making it clear exactly which parts of the code may need to be changed when porting the program.

Some implementation-dependent features may be used to achieve particular performance or efficiency objectives. Commenting these objectives ensures that the programmer can find an appropriate way to achieve them when porting to a different implementation, or explicitly recognize that they cannot be achieved.

Interrupt entries are implementation-dependent features that may not be supported (e.g., VAX Ada uses pragmas to assign system traps to "normal" rendezvous). However, interrupt entries cannot be avoided in most embedded real-time systems and it is reasonable to assume that they are supported by an Ada implementation. The actual value for an interrupt is implementation-defined. Isolate it.

note

Ada can be used to write machine-dependent programs that take advantage of an implementation in a manner consistent with the Ada model, but which make particular choices where Ada allows implementation freedom. These machine dependencies should be treated in the same way as any other implementation- dependent features of the code.

7.1.6 Incorrect Order Dependencies

guideline

- Avoid depending on the order in which certain constructs in Ada are evaluated (see Department of Defense 1983, I-17).

example

The following example intentionally violates some of our guidelines, including naming, use of nonlocal variables, and side-effects. The important thing here is that the commented line depends on Y being evaluated before SQUARE (Y).

```
X, Y : REAL;
...
-------------------------------------------------------------------------
function SQUARE (VALUE : in REAL) return REAL is
begin
   Y := VALUE * VALUE;
   return Y;
end SQUARE;
-------------------------------------------------------------------------
...
X := Y + SQUARE (Y); -- sum Y and its square; make Y contain square of
                     -- its former self; keep the sum in X.
```

rationale

An incorrect order dependency may arise whenever as stated in the Ada Language Reference Manual ". . . specifies that different parts of a given construct are to be executed in some order that is not specified by the language. The construct is incorrect if execution of these parts in a different order would have a different effect" 1 (Department of Defense 1983, §1.6).

While an incorrect order dependency may not adversely effect the program on a certain implementation, the code might not execute correctly when it is ported. Avoid incorrect order dependencies, but also recognize that even an unintentional error of this kind could prohibit portability.

7.2 NUMERIC TYPES AND EXPRESSIONS

A great deal of care was taken with the design of the Ada features related to numeric computations to ensure that the language could be used in embedded systems and mathematical applications where precision was important. As far as possible, these features were made portable. However, there is an inevitable tradeoff between maximally exploiting the available precision of numeric computation on a particular machine and maximizing the portability of Ada numeric constructs. This means that these Ada features, particularly numeric types and expressions, must be used with great care if full portability of the resulting program is to be guaranteed.

7.2.1 Predefined Numeric Types

guideline

- Do not use the predefined numeric types in package STANDARD. Use range and digits declarations and let the implementation do the derivation implicitly from the predefined types.
- For programs that require greater accuracy than that provided by the global assumptions, define a package that declares a private type and operations as needed (see Pappas 1985) for a full explanation and examples.

example

The second example below is not representable as a subrange of INTEGER on a machine with an 8-bit word. The first example below allows a compiler to choose a multiword representation if necessary.

```
-- use
type DAY_OF_LEAP_YEAR is range 1 .. 366;
-- rather than
type DAY_OF_LEAP_YEAR is new INTEGER range 1 .. 366;
```

rationale

An implementor is free to define the range of the predefined numeric types. Porting code from an implementation with greater accuracy to one of lesser is a time consuming and error-prone process. Many of the errors are not reported until run-time.

This applies to more than just numerical computation. An easy-to-overlook instance of this problem occurs if you neglect to use explicitly declared types for integer discrete ranges (array sizes, loop ranges,

etc.) (see Guidelines 5.5.1 and 5.5.2). If you do not provide an explicit type when specifying index constraints and other discrete ranges, a predefined integer type is assumed.

exceptions

The private type and related operations approach can incur considerable overhead. Apply alternative techniques (e.g., subtypes) to those portions of a program requiring greater efficiency.

7.2.2 Ada Model

guideline

- Know the Ada model for floating point types and arithmetic.

rationale

Declarations of Ada floating point types give users control over both the representation and arithmetic used in floating point operations. Portable properties of Ada programs are derived from the models for floating point numbers of the subtype and the corresponding safe numbers. The relative spacing and range of values in a type are determined by the declaration. Attributes can be used to specify the transportable properties of an Ada floating point type.

7.2.3 Analysis

guideline

- Carefully analyze what accuracy and precision you really need.

rationale

Floating point calculations are done with the equivalent of the implementation's predefined floating point types. The effect of extra "guard" digits in internal computations can sometimes lower the number of digits that must be specified in an Ada declaration. This may not be consistent over implementations where the program is intended to be run. It may also lead to the false conclusion that the declared types are sufficient for the accuracy required.

The numeric type declarations should be chosen to satisfy the lowest precision (smallest number of digits) that will provide the required accuracy. Careful analysis will be necessary to show that the declarations are adequate.

7.2.4 Accuracy Constraints

guideline

- Do not press the accuracy limits of the machine(s).

rationale

The Ada floating point model is intended to facilitate program portability, which is often at the expense of efficiency in using the underlying machine arithmetic. Just because two different machines use the same number of digits in the mantissa of a floating point number does not imply they will have the same arithmetic properties. Some Ada implementations may give slightly better accuracy than required by Ada because they make efficient use of the machine. Do not write programs that depend on this.

7.2.5 Comments

guideline

- Comment the analysis and derivation of the numerical aspects of a program.

rationale

Decisions and background about why certain precisions are required in a program are important to program revision or porting. The underlying numerical analysis leading to the program should be commented.

7.2.6 Precision of Constants

guideline

- Use named numbers or universal real expressions rather than constants of any particular type.

rationale

For a given radix (number base), there is a loss of accuracy for some rational and all irrational numbers when represented by a finite sequence of digits. Ada has named numbers and expressions of type universal_real that provide maximal accuracy of representation in the source program. These numbers and expressions are converted to finite representations at compile time. By using universal real expressions and numbers, the programmer can automatically delay the conversion to machine types until the point where it can be done with the minimum loss of accuracy.

note

See also Guideline 3.2.5.

7.2.7 Appropriate Radix

guideline

- Represent literals in a radix appropriate to the problem.

example

```
type MAXIMUM_SAMPLES     is range  1           .. 1_000_000;
type LEGAL_HEX_ADDRESS   is range 16#0000#     .. 16#FFFF#;
type LEGAL_OCTAL_ADDRESS is range  8#000_000# .. 8#777_777#;
```

rationale

Ada provides a way of representing numbers using a radix other than ten. These numbers are called based literals (Ada Reference Manual 1983, §2.4.2). The choice of radix determines whether the representation of a radix fraction will terminate or repeat. This technique is appropriate when the problem naturally uses some base other than ten for its numbers.

7.2.8 Subexpression Evaluation

guideline

- Anticipate values of subexpressions to avoid exceeding the range of their type. Use derived types, subtypes, factoring, and range constraints on numeric types as described in Guidelines 3.4.1, 5.3.1, 5.5.3, and 5.5.6.

rationale

The Ada language does not require that an implementation perform range checks on subexpressions within an expression. Even if the implementation on your program's current target does not perform these checks, your program may be ported to an implementation that does.

7.2.9 Relational Tests

guideline

- Do relational tests with `<=` and `>=` rather than `<`, `>`, `=`, and `/=`.

rationale

Strict relational comparisons (`<`, `>`, `=`, `/=`) are a general problem in floating point computations. Because of the way Ada comparisons are defined in terms of model intervals, it is possible for the values of the Ada comparisons `A < B` and `A = B` to depend on the implementation, while `A <= B` evaluates uniformly across implementations. Note that for floating point in Ada, `"A <= B"` is not the same as `"not (A > B)"`.

7.2.10 Type Attributes

guideline

- Use values of type attributes in comparisons and checking for small values.

example

The following examples test for (1) absolute "equality" in storage, (2) absolute "equality" in computation, (3) relative "equality" in storage, and (4) relative "equality" in computation.

```
if abs (X - Y) <= FLOAT_TYPE'SMALL                -- (1)
if abs (X - Y) <= FLOAT_TYPE'BASE'SMALL           -- (2)
if abs (X - Y) <= abs X * FLOAT_TYPE'EPSILON      -- (3)
if abs (X - Y) <= abs X * FLOAT_TYPE'BASE'EPSILON -- (4)
```

rationale

These attributes are the primary means of symbolically accessing the implementation of the Ada numeric model. When the characteristics of the model numbers are accessed symbolically, the source code is portable. The appropriate model numbers of any implementation will then be used by the generated code.

7.2.11 Testing Special Operands

guideline

- Test carefully around special values.

rationale

Tests around zero are particularly troublesome; for example, if X is any value mathematically in the range -T'SMALL < X < T'SMALL, it is possible for either (and maybe both) of the Ada expressions X <= 0.0 or X >= 0.0 to evaluate to TRUE.

7.3 STORAGE CONTROL

The management of dynamic storage can vary between Ada environments. In fact, some environments do not provide any deallocation. The guidelines in this section encourage the programmer to bring dynamic storage management under explicit program control to improve the portability of programs using it.

7.3.1 Collection Size for Access Types

guideline

- Use a representation clause to specify the collection size for access types. Specify the collection size in general terms using the 'SIZE attribute of the object type.

example

```
type PERSONNEL_INFORMATION is
  record
    -- desired information
  end PERSONNEL_INFORMATION;

type SUBJECT_EMPLOYEE is access PERSONNEL_INFORMATION;

for SUBJECT_EMPLOYEE'STORAGE_SIZE use
  (NUMBER_OF_EMPLOYEES + SLACK)
    * (PERSONNEL_INFORMATION'SIZE / SYSTEM.STORAGE_UNIT);
```

rationale

There are many variations among implementations of dynamic storage algorithms. Here is a brief summary of some of the issues:

- The processing time to acquire the storage and then later free it up (with possible garbage collection) can vary greatly.
- The time at which overhead is incurred (e.g., obtaining a pool at type declaration time versus individual objects when created versus seemingly random garbage collection) varies greatly.

- The total amount of space available to a given scope may be restricted.
- Dynamic storage pools, with Ada runtime implementations that employ them, may be shared among unconstrained arrays, records with discriminants, and miscellaneous run-time data structures.

Given this degree of variability, it is advantageous to use a representation clause to specify the exact requirements for a given type even though the representation clause is itself an implementation-dependent feature.

note

The amount of storage specified using the representation clause need not be static.

Some implementations give you a fewer number of objects than requested, due to allocation scheme overhead. Be certain to provide allowance for this possibility.

7.3.2 Task Storage

guideline

- Use a representation clause to identify the expected stack space requirements for each task.

rationale

Implementations may vary greatly in the manner in which task stack space is obtained. The varying methods may affect performance or access type storage allocation (when stack space is obtained from heaps).

Even though a representation clause is an optional and implementation dependent feature (in the worst case it will be ignored), it provides a mechanism for control of dynamic memory allocation with respect to task activation.

7.4 TASKING

The definition of tasking in the Ada language leaves many characteristics of the tasking model up to the implementor. This allows a vendor to make appropriate tradeoffs for the intended application domain, but it also diminishes the portability of designs and code employing the tasking features. In some respects this diminished portability is an inherent characteristic of concurrency approaches (see Nissen and Wallis 1984, 37).

A discussion of Ada tasking dependencies when employed in a distributed target environment is beyond the scope of this book. For example, multi-processor task scheduling, interprocessor rendezvous, and the distributed sense of time through package CALENDAR are all subject to differences between implementations. For more information, (Nissen and Wallis 1984 and ARTEWG 1986) touch on these issues and (Volz et al. 1985) is one of many research articles available.

7.4.1 Task Activation Order

guideline

- Do not depend on the order in which task objects are activated when declared in the same declarative list.

rationale

The order is left undefined in the Ada LRM (Ada Reference Manual 1983).

7.4.2 Delay Statements

guideline

- Do not depend on a particular delay being achievable (Nissen and Wallis 1984).
- Never use a busy waiting loop instead of a delay.
- Design to limit polling to those cases where it is absolutely necessary.
- Never use knowledge of the execution pattern of tasks to achieve timing requirements.

rationale

The rationale for this appears in Guideline 6.1.5. In addition, the treatment of delay statements varies from implementation to implementation thereby hindering portability.

7.4.3 Package CALENDAR, Type DURATION, and SYSTEM.TICK

guideline

- Do not assume a correlation between SYSTEM.TICK and package CALENDAR or type DURATION (see Guideline 6.1.5).

rationale

Such a correlation is not required, although it may exist in some implementations.

7.4.4 Select Statement Evaluation Order

guideline

- Do not depend on the order in which guard conditions are evaluated or on the algorithm for choosing among several open select alternatives.

rationale

The language does not define the order of these conditions, so assume that they are arbitrary.

7.4.5 Task Scheduling Algorithm

guideline

- Do not assume that tasks execute uninterrupted until they reach a synchronization point.
- Use pragma PRIORITY to distinguish general levels of importance only (see Guideline 6.1.4).

rationale

The Ada tasking model requires that tasks be synchronized only through the explicit means provided in the language (i.e., rendezvous, task dependence, pragma SHARED). The scheduling algorithm is not defined by the language and may vary from time sliced to preemptive priority. Some implementations (e.g., VAX Ada) provide several choices that a user may select for the application.

note

The number of priorities may vary between implementations. In addition, the manner in which tasks of the same priority are handled may vary between implementations even if the implementations use the same general scheduling algorithm.

exceptions

In real-time systems it is often necessary to tightly control the tasking algorithm to obtain the required performance. For example, avionics systems are frequently driven by cyclic events with limited asynchronous interruptions. A nonpreemptive tasking model is traditionally used to obtain the greatest performance in these applications. Cyclic executives can be programmed in Ada, as can a progression of scheduling schemes from cyclic through multiple-frame-rate to full asynchrony (MacLaren 1980) although an external clock is usually required.

7.4.6 Abort

guideline

- Avoid using the abort statement.

rationale

The rationale for this appears in Guideline 6.3.3. In addition, treatment of the abort statement varies from implementation to implementation thereby hindering portability.

7.4.7 Shared Variables and Pragma SHARED

guideline

- Do not share variables.
- Have tasks communicate through the rendezvous mechanism.
- Do not use shared variables as a task synchronization device.
- Use pragma SHARED only when you are forced to by run time system deficiencies.

rationale

The rationale for this appears in Guideline 6.2.4. In addition, the treatment of shared variables varies from implementation to implementation thereby hindering portability.

7.5 EXCEPTIONS

Care must be exercised using predefined exceptions as aspects of their treatment may vary between implementations. Implementation-defined exceptions must, of course, be avoided.

7.5.1 Predefined Exceptions

guideline

- Do not depend on the exact locations at which predefined exceptions are raised.

rationale

The Ada Language Reference Manual (Department of Defense 1983) states that among implementations, a predefined exception for the same cause may be raised from different locations. You will not be able to discriminate between the exceptions. Further, each of the predefined exceptions is associated with a variety of conditions. Any exception handler written for a predefined exception must be prepared to deal with any of these conditions.

7.5.2 CONSTRAINT_ERROR and NUMERIC_ERROR

guideline

- Program with the possibility of CONSTRAINT_ERROR as well as NUMERIC_ERROR.

rationale

Either of these exceptions may be raised (and different implementations may raise either one under otherwise similar circumstances). Exception handlers should be prepared to handle either.

7.5.3 Implementation-Defined Exceptions

guideline

- Do not raise implementation-defined exceptions.

rationale

No exception defined by an implementation can be guaranteed to be portable to other implementations whether or not they are from the same vendor. Not only may the names be different, but the range of conditions triggering the exceptions may be different also.

exceptions

If you create interface packages for the implementation-specific portions of your program, you can have those packages "export" the implementation-defined exceptions, or better, define user exceptions. Keep the names you use for these general. Do not allow yourself to be forced to find and change the name of every handler you have written for these exceptions when the program is ported.

7.6 REPRESENTATION CLAUSES AND IMPLEMENTATION-DEPENDENT FEATURES

Ada provides many implementation-dependent features that permit greater control over and interaction with the underlying hardware architecture than is normally provided by a high-order language. These mechanisms are intended to assist in systems programming and real-time programming to obtain greater efficiency (e.g., specific size layout of variables through representation clauses) and direct hardware interaction (e.g., interrupt entries) without having to resort to assembly level programming.

Given the objectives for these features, it is not surprising that you must usually pay a significant price in portability to use them. In general, where portability is the main objective, do not use these features. When you must use these features, encapsulate them in packages well-commented as interfacing to the particular target environment. This section identifies the various features and their recommended use with respect to portability.

7.6.1 Representation Clauses

guideline

- Isolate the use of representation clauses.

rationale

The Ada LRM (Ada Reference Manual 1983) does not require that these clauses be supported for all types. Therefore, isolating representation clauses will minimize the impact of any changes necessitated by a port.

exceptions

The two exceptions to this guideline are for task storage size and access collection size, where portability may be enhanced through their use (see Guidelines 7.3.1 and 7.3.2).

7.6.2 Package SYSTEM

guideline

- Avoid using package SYSTEM constants except in attempting to generalize other machine dependent constructs.

rationale

Since the values in this package are implementation-provided, unexpected effects can result from their use.

exceptions

Do use package SYSTEM constants to parameterize other implementation-dependent features (see [Pappas 1985] examples for numeric ranges [§13.7.1] and access collection size [§4.8]).

7.6.3 Machine Code Inserts

guideline

- Avoid machine code inserts.

rationale

There is no requirement that this feature be implemented. It is possible that two different vendors' syntax will differ for an identical target and certainly, differences in lower-level details such as register conventions will hinder portability.

exceptions

If machine code inserts must be used to meet another project requirement, recognize the portability decreasing effects and isolate and highlight their use.

In the commentary include that a machine code insert is being used, what function the insert provides, and (especially) why the insert is necessary. Comment the necessity of using machine code inserts by delineating what went wrong with attempts to use other higher-level constructs.

7.6.4 Interfacing Foreign Languages

guideline

- Avoid interfacing Ada with other languages.
- Isolate all subprograms employing pragma INTERFACE to an implementation-dependent (interface) package.

rationale

The problems with employing pragma INTERFACE are complex. These problems include pragma syntax differences, conventions for linking/binding Ada to other languages, and mapping Ada variables to foreign language variables.

exceptions

It is often necessary to interact with other languages, if only an assembly language to reach certain hardware features. In these cases, clearly comment the requirements and limitations of the interface and pragma INTERFACE usage.

7.6.5 Implementation-Defined Pragmas and Attributes

guideline

- Avoid pragmas and attributes added by the implementor.

rationale

The Ada LRM (Ada Reference Manual 1983) permits an implementor to add pragmas and attributes to exploit a particular hardware architecture or software environment. These are obviously even more implementation-specific and therefore less portable than an implementor's interpretations of the predefined pragmas and attributes.

exceptions

Some implementation-dependent features are gaining wide acceptance in the Ada community to help alleviate inherent inefficiencies in some Ada features. A good example of this is the "fast interrupt" mechanism that provides a minimal interrupt latency time in exchange for a restrictive tasking environment. Ada community groups (e.g., SIGAda's ARTEWG) are attempting to standardize a common mechanism and syntax to provide this capability. By being aware of industry trends when specialized features must be used, you can take a more general approach that will help minimize the porting job.

7.6.6 Unchecked Deallocation

guideline

- Avoid dependence on UNCHECKED_DEALLOCATION (see Guideline 5.9.2).

rationale

The unchecked storage deallocation mechanism is one method for overriding the default time at which allocated storage is reclaimed. The earliest default time is when an object is no longer accessible, e.g., when control leaves the scope where an access type was declared (the exact point after this time is implementation- dependent). Any unchecked deallocation of storage performed prior to this may result in an erroneous Ada program if an attempt is made to access the object.

This guideline is stronger than Guideline 5.9.2 because of the extreme dependence on the implementation of UNCHECKED_DEALLOCATION. Using it could cause considerable difficulty with portability.

exceptions

Using unchecked deallocation of storage can be beneficial in local control of highly iterative or recursive algorithms where available storage may be exceeded. Be careful to avoid erroneous situations as described above.

7.6.7 Unchecked Conversion

guideline

- Avoid using UNCHECKED_CONVERSION (see Guideline 5.9.1).

rationale

The unchecked type conversion mechanism is, in effect, a means of bypassing the strong typing facilities in Ada. An implementation is free to limit the types that may be matched and the results that occur when object sizes differ.

exceptions

Unchecked type conversion is useful in implementation dependent parts of Ada programs (where lack of portability is isolated) where low-level programming and foreign language interfacing is the objective.

7.6.8 Run Time Dependencies

guideline

- Avoid the direct invocation of or implicit dependence upon an underlying host operating system or Ada run time support system.

rationale

Features of an implementation not specified in the Ada LRM (Ada Reference Manual 1983) will usually differ between implementations. Specific implementation-dependent features are not likely to be provided in other implementations. Even if a majority of vendors eventually provide similar features, they are unlikely to have identical formulations. Indeed, different vendors may use the same formulation for (semantically) entirely different features.

Try to avoid these when coding. Consider the consequences of including system calls in a program on a host development system. If these calls are not flagged for removal and replacement, the program could go through development and testing only to be unusable when moved to a target environment which lacks the facilities provided by those system calls on the host.

exceptions

In real-time embedded systems, making calls to low-level support system facilities may often be unavoidable. Isolate the uses of these facilities may often be unavoidable. Comment them as you would machine code inserts (see Guideline 7.6.3); they are, in a sense, instructions for the virtual machine provided by the support system. When isolating the uses of these features, provide an interface for the rest of your program to use which can be ported through replacement of the interface's implementation.

7.6.9 System Partitioning

guideline

- Minimize artificial partitioning of an Ada program to exploit specific architectures.

examples

Example architectures with small address spaces include many of the 16-bit architectures such as the MIL-STD-1750A or Intel 8086/80186 (where only 128K bytes of the 1 to 2M bytes is directly addressable) or the U.S. Navy AN/UYK-44 or AN/AYK-14 (where only 64K bytes of the 2 to 4M bytes is directly addressable).

rationale

For applications whose size exceeds that of the direct address space of the target architecture, it is often necessary for an Ada implementation to force a partitioning that is unnatural to the Ada style (e.g., limited use of context clauses and generic invocation).

exceptions

If a limited address space target must be used, performance considerations may force artificial partitioning.

7.7 INPUT/OUTPUT

The I/O facilities in Ada are not a part of the syntactic definition of the language. The constructs in the language have been used to define a set of packages for this purpose. These packages are not expected to meet all the I/O needs of all applications, in particular embedded systems. They serve as a core subset that may be used on straightforward data, and that can be used as examples of building I/O facilities upon the low-level constructs provided by the language. Providing an I/O definition that could meet the requirements of all applications and integrate with the many existing operating systems would result in unacceptable implementation dependencies.

The types of portability problems encountered with I/O tend to be different for applications running with a host operating system versus embedded targets where the Ada run-time is self-sufficient. Interacting with a host operating system offers the added complexity of coexisting with the host file system structures (e.g., hierarchical directories), access methods (e.g., ISAM) and naming conventions (e.g., logical names and aliases based on the current directory). The section on I/O in (ARTEWG 1986) provides some examples of this type of dependency. Embedded applications have different dependencies that often tie them to the low-level details of their hardware devices.

The major defense against these inherent implementation dependencies in I/O is to try to isolate their functionality in any given application. The majority of the following guidelines are focused in this direction.

7.7.1 Implementation-Added Features

guideline

- Avoid the use of additional I/O features provided by a particular vendor.

rationale

Vendor-added features are not likely to be provided by other implementations. Even if a majority of vendors eventually provide similar additional features, they are unlikely to have identical formulations. Indeed, different vendors may use the same formulation for (semantically) entirely different features.

exceptions

There are many types of applications that require the use of these features. Examples include: multilingual systems that standardize on a vendor's file system, applications that are closely integrated with vendor products (i.e., user interfaces), and embedded systems for performance reasons. Isolate the use of these features into packages.

7.7.2 NAME and FORM Parameters

guideline

- Use constants and variables as symbolic actuals for the NAME and FORM parameters on the predefined I/O packages. Declare and initialize them in an implementation dependency package.

rationale

The format and allowable values of these parameters on the predefined I/O packages can vary greatly between implementations. Isolation of these values facilitates portability. Note that not specifying a FORM string or using a null value does not guarantee portability since the implementation is free to specify defaults.

note

It may be desirable to further abstract the I/O facilities by defining additional CREATE and OPEN procedures that hide the visibility of the FORM parameter entirely see (Pappas 1985, 54-55).

7.7.3 File Closing

guideline

- Close all files explicitly.

rationale

The Ada LRM (Ada Reference Manual 1983, §14.1) states, "The language does not define what happens to external files after completion of the main program (in particular, if corresponding files have not been closed)." The possibilities range from being closed in an anticipated manner to deletion.

The disposition of a closed temporary file may vary, perhaps affecting performance and space availability (ARTEWG 1986).

7.7.4 I/O on Access Types

guideline

- Avoid performing I/O on access types.

rationale

The Ada LRM (Ada Reference Manual 1983) does not require that it be supported. When such a value is written, it is placed out of reach of the implementation. Thus, it is out of reach of the reliability-enhancing controls of strong type checking.

Consider the meaning of this operation. One possible implementation of the values of access types is virtual addresses. If you write such a value, how can you expect another program to read that value and make any sensible use of it? The value cannot be construed to refer to any meaningful location within the reader's address space, nor can a reader infer any information about the writer's address space from the value read. The latter is the same problem that the writer would have trying to interpret or use the value if it is read back in. To wit, a garbage collection and/or heap compaction scheme may have moved the item formerly accessed by that value, leaving that value "pointing" at space which is now being put to indeterminable uses by the underlying implementation.

7.7.5 Package LOW_LEVEL_IO

guideline

- Minimize and isolate using the predefined package LOW_LEVEL_IO.

rationale

LOW_LEVEL_IO is intended to support direct interaction with physical devices that are usually unique to a given host or target environment. In addition, the data types provided to the procedures are implementation-defined. This allows vendors to define different interfaces to an identical device.

exceptions

Those portions of an application that must deal with this level of I/O, e.g., device drivers and real-time components dealing with discretes, are inherently nonportable. Where performance allows, structure these components to isolate the hardware interface. Only within these isolated portions is it advantageous to employ the LOW_LEVEL_IO interface which is portable in concept and general procedural interface, if not completely so in syntax and semantics.

7.8 SUMMARY

fundamentals

- Make considered assumptions about the support provided for the following on potential target platforms:
- Determine the actual properties and limits of the Ada implementation(s) you are using.
- Use highlighting comments for each package, subprogram and task where any nonportable features are present.
- For each nonportable feature employed, describe the expectations for that feature.
- Avoid using any implementation features associated with the main subprogram (e.g., allowing parameters to be passed).
- Encapsulate hardware and implementation dependencies in a package.

- Clearly indicate the objectives if machine or solution efficiency is the reason for hardware or implementation dependent code.
- Develop specific bodies for specific applications to meet particular needs or constraints after porting.
- Isolate interrupt receiving tasks into implementation dependent packages.
- Avoid depending on the order in which certain constructs in Ada are evaluated.

numeric types and expressions

- Do not use the predefined numeric types in package STANDARD. Use range and digits declarations and let the implementation do the derivation implicitly from the predefined types.
- For programs that require greater accuracy than that provided by the global assumptions, define a package that declares a private type and operations as needed.
- Know the Ada model for floating point types and arithmetic.
- Carefully analyze what accuracy and precision you really need.
- Do not press the accuracy limits of the machine(s).
- Comment the analysis and derivation of the numerical aspects of a program.
- Use named numbers or universal real expressions rather than constants of any particular type.
- Represent literals in a radix appropriate to the problem.
- Anticipate values of subexpressions to avoid exceeding the range of their type. Use derived types, subtypes, factoring, and range constraints on numeric types as described in Guidelines 3.4.1, 5.3.1, 5.5.3, and 5.5.6.
- Do relational tests with <= and >= rather than <, >, =, and /=.
- Use values of type attributes in comparisons and checking for small values.
- Test carefully around special values.

storage control

- Use a representation clause to specify the collection size for access types. Specify the collection size in general terms using the 'SIZE attribute of the object type.
- Use a representation clause to identify the expected stack space requirements for each task.

tasking

- Do not depend on the order in which task objects are activated when declared in the same declarative list.
- Do not depend on a particular delay being achievable.
- Never use a busy waiting loop instead of a delay.
- Design to limit polling to those cases where it is absolutely necessary.
- Never use knowledge of the execution pattern of tasks to achieve timing requirements.
- Do not assume a correlation between SYSTEM.TICK and package CALENDAR or type DURATION (see Guideline 6.1.5).
- Do not depend on the order in which guard conditions are evaluated or on the algorithm for choosing among several open select alternatives.
- Do not assume that tasks execute uninterrupted until they reach a synchronization point.
- Use pragma PRIORITY to distinguish general levels of importance only (see Guideline 6.1.4).
- Avoid using the abort statement.
- Do not share variables.
- Have tasks communicate through the rendezvous mechanism.
- Do not use shared variables as a task synchronization device.

- Use pragma SHARED only when you are forced to by run time system deficiencies.

exceptions

- Do not depend on the exact locations at which predefined exceptions are raised.
- Program with the possibility of CONSTRAINT_ERROR as well as NUMERIC_ERROR.
- Do not raise implementation-defined exceptions.

representation clauses and implementation-dependent features

- Isolate the use of representation clauses.
- Avoid using of package SYSTEM constants except in attempting to generalize other machine dependent constructs.
- Avoid machine code inserts.
- Avoid interfacing Ada with other languages.
- Isolate all subprograms employing pragma INTERFACE to an implementation-dependent (interface) package.
- Avoid pragmas and attributes added by the implementor.
- Avoid dependence on UNCHECKED_DEALLOCATION (see Guideline 5.9.2).
- Avoid using UNCHECKED_CONVERSION (see Guideline 5.9.1).
- Avoid the direct invocation of or implicit dependence upon an underlying host operating system or Ada run time support system.
- Minimize artificial partitioning of an Ada program to exploit specific architectures.

input/output

- Avoid the use of additional I/O features provided by a particular vendor.
- Use constants and variables as symbolic actuals for the NAME and FORM parameters on the predefined I/O packages. Declare and initialize them in an implementation dependency package.
- Close all files explicitly.
- Avoid performing I/O on access types.
- Minimize and isolate using the predefined package LOW_LEVEL_IO.

CHAPTER 8
Reusability

There are many issues involved in software reuse, including whether to reuse parts, how to store and retrieve reusable parts in a library, how to certify parts, how to maximize the economic value of reuse, how to provide incentives to engineers and entire companies to reuse parts rather than reinvent them, and so on. This chapter ignores these managerial, economic, and logistic issues to focus on the single technical issue of how to write software parts in Ada to increase reuse potential. The other issues are just as important but are outside of the scope of this book.

One of the design goals of Ada was to facilitate the creation and use of reusable parts to improve productivity. To this end, Ada provides features to develop reusable parts and to adapt them once they are available. Packages, visibility control, and separate compilation support modularity and information hiding (see Guidelines in Sections 4.1, 4.2, 5.3 and 5.7). This allows the separation of application-specific parts of the code, maximizing the general purpose parts suitable for reuse, and allows the isolation of design decisions within modules, facilitating change. The Ada type system supports localization of data definitio so that consistent changes are easy to make. Generic units directly support the development of gener... purpose, adaptable code that can be instantiated to perform specific functions. Using these features carefully, and in conformance to the guidelines in this book, produces code that is more likely to be reusable.

The guidelines in this chapter are concerned with how to write reusable Ada code. The underlying assumption is that reusable parts are rarely built in isolation and are hard to recover from code that was developed without reuse in mind. The guidelines therefore focus on how to produce reusable parts as a by-product of developing software for specific applications.

A reusable part must fulfill a number of different criteria. This chapter is organized around the following criteria:

- Reusable parts must be understandable. A reusable part should be a model of clarity. The requirements for commenting reusable parts are even more stringent than those for parts specific to a particular application.
- Reusable parts must be of the highest possible quality. They must be correct, reliable, and robust. An error or weakness in a reusable part may have far-reaching consequences, and it is important that other programmers can have a high degree of confidence in any parts offered for reuse.
- Reusable parts must be adaptable. To maximize its reuse potential, a part must be able to adapt to the needs of a wide variety of users.
- Reusable parts should be independent. It should be possible to reuse a single part without also adopting many other parts that are apparently unrelated.

In addition to these criteria, a reusable part must be easier to reuse than to reinvent, must be efficient, and must be portable. If it takes more effort to reuse a part than to create one from scratch, or if the reused part is simply not efficient enough, reuse does not occur as readily. For guidelines on portability, see Chapter 7.

This chapter should not be read in isolation. In many respects, a well-written, reusable component is simply an extreme example of a well-written component. All of the guidelines in the previous chapters apply to

reusable components as well as components specific to a single application. The guidelines listed here apply specifically to reusable components.

8.1 UNDERSTANDING AND CLARITY

It is particularly important that parts intended for reuse should be easy to understand. The following must be immediately apparent from inspection of the comments and the code itself: what the part does, how to use it, what anticipated changes might be made to it in the future, and how it works. For maximum readability of reusable parts, follow the guidelines in Chapter 3, some of which are repeated more strongly below.

8.1.1 Application-Independent Naming

guideline

- Select the least restrictive names possible for reusable parts and their identifiers.
- Reserve the best name for a generic instantiation, using the second best for the generic unit itself.
- Use names which indicate the behavioral characteristics of the reusable part, as well as its abstraction.

example

General-purpose stack abstraction:

```
-------------------------------------------------------------------------
generic
   type ITEM is limited private;
   ...
package GENERIC_BOUNDED_STACK is
   procedure PUSH (...);
   procedure POP (...);
   ...
end GENERIC_BOUNDED_STACK;
-------------------------------------------------------------------------
```

Renamed appropriately for use in current application:

```
-------------------------------------------------------------------------
with GENERIC_BOUNDED_STACK;
package CAFETERIA is
   type TRAYS is ...;
   package TRAY_STACK is new GENERIC_BOUNDED_STACK (ITEM => TRAYS, ...);
   ...
end CAFETERIA;
-------------------------------------------------------------------------
```

rationale

Choosing a general or application-independent name for a reusable part encourages its wide reuse. When the part is used in a specific context, it can be instantiated (if generic) or renamed with a more specific name.

When there is an obvious choice for the simplest, clearest name for a reusable part, it is a good idea to leave that name for use by the reuser of the part, choosing a longer, more descriptive name for the reusable part. Thus, GENERIC_BOUNDED_STACK is a better name than STACK for a generic stack package because it leaves the simpler name STACK available to be used by an instantiation.

Include indications of the behavioral characteristics (but not indications of the implementation) in the name of a reusable part so that multiple parts with the same abstraction (e.g., multiple stack packages) but with different restrictions (bounded, unbounded, etc.) can be stored in the same Ada library and used as part of the same Ada program.

8.1.2 Abbreviations

guideline

- Do not use any abbreviations in identifier or unit names.

rationale

This is a stronger guideline than Guideline 3.1.4. However well commented, an abbreviation may cause confusion in some future reuse context. Even universally accepted abbreviations, such as GMT for Greenwich Mean Time, can cause problems and should be used only with great caution.

note

When reusing a part in a specific application, consider renaming the part using abbreviations standard to that application.

8.2 ROBUSTNESS

The guidelines below improve the robustness of Ada code. It is easy to write code that depends on an assumption which you do not realize that you are making. When such a part is reused in a different environment, it can break unexpectedly. The guidelines below show some ways in which Ada code can be made to automatically conform to its environment, and some ways in which it can be made to check for violations of assumptions. Finally, some guidelines are given to warn you about errors which Ada does not catch as soon as you might like.

8.2.1 Symbolic Constants

guideline

- Use symbolic constants and constant expressions to allow multiple dependencies to be linked to a small number of symbols.

example

```
procedure DISK_DRIVER is
   --In this procedure, a number of important disk parameters are linked.
   NUMBER_SECTORS       : constant := 4;
   NUMBER_TRACKS        : constant := 200;
   NUMBER_SURFACES      : constant := 18;
   SECTOR_CAPACITY      : constant := 4096;
   TRACK_CAPACITY       : constant := NUMBER_SECTORS  * SECTOR_CAPACITY;
   SURFACE_CAPACITY     : constant := NUMBER_TRACKS   * TRACK_CAPACITY;
   DISK_CAPACITY        : constant := NUMBER_SURFACES * SURFACE_CAPACITY;

   type SECTOR_RANGE    is range 1 .. NUMBER_SECTORS;
   type TRACK_RANGE     is range 1 .. NUMBER_TRACKS;
   type SURFACE_RANGE   is range 1 .. NUMBER_SURFACES;

   type TRACK_MAP       is array (SECTOR_RANGE)  of ...;
   type SURFACE_MAP     is array (TRACK_RANGE)   of TRACK_MAP;
   type DISK_MAP        is array (SURFACE_RANGE) of SURFACE_MAP;

begin
   ...
end DISK_DRIVER;
```

rationale

To reuse software that uses symbolic constants and constant expressions appropriately, just one or a small number of constants need to be reset and all declarations and associated code are changed automatically. Apart from easing reuse, this reduces the number of opportunities for error and documents the meanings of the types and constants without using error-prone comments.

8.2.2 Unconstrained Arrays

guideline

- Use unconstrained array types for array formal parameters and array return values.
- Make the size of local variables depend on actual parameter size where appropriate.

example

```
type VECTOR is
  array (VECTOR_INDEX range <>) of ELEMENT;
type MATRIX is
  array (VECTOR_INDEX range <>, VECTOR_INDEX range <>) of ELEMENT;
...
------------------------------------------------------------------------
procedure MATRIX_OPERATION (DATA : in MATRIX) is
   WORKSPACE   : MATRIX (DATA'RANGE (1), DATA'RANGE (2));
   TEMP_VECTOR : VECTOR (DATA'FIRST (1) .. 2 * DATA'LAST (1));
```

rationale

Unconstrained arrays can be declared with their sizes dependent on formal parameter sizes. When used as local variables, their sizes change automatically with the supplied actual parameters. This facility can be used to assist in the adaption of a part since necessary size changes in local variables are taken care of automatically.

8.2.3 Assumptions

guideline

- Minimize the number of assumptions made by a unit.
- For assumptions which cannot be avoided, use types to automatically enforce conformance.
- For assumptions which cannot be automatically enforced by types, add explicit checks to the code.
- Document all assumptions.

example

The following poorly written function documents, but does not check, its assumption:

```
------------------------------------------------------------------------
-- Assumption:  BCD value is less than 4 digits.
------------------------------------------------------------------------
function BINARY_TO_BCD (BINARY_VALUE : in NATURAL) return ... is
begin
   ...
end BINARY_TO_BCD;
------------------------------------------------------------------------
```

The next example documents and explicitly checks its assumption:

```
------------------------------------------------------------------------
-- Exceptions:  OUT_OF_RANGE raised when BCD value exceeds 4 digits.
------------------------------------------------------------------------
function BINARY_TO_BCD (BINARY_VALUE : in NATURAL) return ... is
   MAX_REPRESENTABLE : constant NATURAL := 999;
begin
   if BINARY_VALUE > MAX_REPRESENTABLE then
      raise OUT_OF_RANGE;
   end if;
   ...
end BINARY_TO_BCD;
------------------------------------------------------------------------
```

The last example enforces conformance with its assumption, making the checking automatic, and the comment unnecessary:

```
type BINARY_VALUES is new NATURAL range 0 .. 999;
------------------------------------------------------------------------
function BINARY_TO_BCD (BINARY_VALUE : in BINARY_VALUES) return ... is
begin
   ...
end BINARY_TO_BCD;
------------------------------------------------------------------------
```

rationale

Any part that is intended to be used again in another program, especially if the other program is likely to be written by other people, should be robust. It should defend itself against misuse by defining its

interface to enforce as many assumptions as possible and by adding explicit defensive checks on anything which cannot be enforced by the interface.

note

You can restrict the ranges of values of the inputs by careful selection or construction of the types of the formal parameters. When you do so, the compiler-generated checking code may be more efficient than any checks you might write. Indeed, such checking is part of the intent of the strong typing in the language. This presents a challenge, however, for generic units where the user of your code selects the types of the parameters. Your code must be constructed so as to deal with any value of any type the user may choose to select for an instantiation.

8.2.4 Subtypes in Generic Specifications

guideline

- Beware of using subtypes as type marks when declaring generic formal objects of type `in out`.
- Beware of using subtypes as type marks when declaring parameters or return values of generic formal subprograms.
- Use symbolic expressions of attributes rather than literal values in reference to generic formal objects, and parameter and return values of generic formal subprograms.

example

In the following example, it appears that any value supplied for the generic formal object `OBJECT` would be constrained to the range `1..10`. It also appears that parameters passed at run-time to the PUT routine in any instantiation, and values returned by the GET routine, would be similarly constrained.

```
subtype RANGE_1_10 is integer range 1 .. 10;
-------------------------------------------------------------------
generic
   OBJECT : in out RANGE_1_10;
   with procedure PUT (PARAMETER : in RANGE_1_10);
   with function  GET return RANGE_1_10;
package INPUT_OUTPUT is
   ...
end INPUT_OUTPUT;
-------------------------------------------------------------------
```

However, this is not the case. Given the following legal instantiation:

```
subtype RANGE_15_30 is integer range 15 .. 30;
CONSTRAINED_OBJECT : RANGE_15_30 := 15;
-------------------------------------------------------------------
procedure CONSTRAINED_PUT (PARAMETER : in RANGE_15_30);
-------------------------------------------------------------------
function  CONSTRAINED_GET return RANGE_15_30;
-------------------------------------------------------------------
package CONSTRAINED_INPUT_OUTPUT is
  new INPUT_OUTPUT (OBJECT => CONSTRAINED_OBJECT,
                    PUT    => CONSTRAINED_PUT,
                    GET    => CONSTRAINED_GET);
-------------------------------------------------------------------
```

`OBJECT`, `PARAMETER`, and the return value of `GET` are constrained to the range `15..30`. Thus, for example, if the body of the generic package contains an assignment statement:

```
OBJECT := 1;
```

`CONSTRAINT_ERROR` is raised when this instantiation is executed.

rationale

According to sections 12.1.1(5) and 12.1.3(5) of the Ada Language Reference Manual (Department of Defense 1983), when constraint checking is performed for generic formal objects, and parameters and return values of generic formal subprograms, the constraints of the actual subtype (not the formal subtype or the base type) are enforced.

Thus, even with a generic unit which has been instantiated and tested many times, and with an instantiation which reported no errors at instantiation time, there can be a run-time error. Since the subtype constraints of the generic formal are ignored, the Ada Language Reference Manual

(Department of Defense 1983) suggests using the name of a base type in such places to avoid confusion. Even so, you must be careful not to assume the freedom to use any value of the base type because the instantiation imposes the subtype constraints of the generic actual parameter. To be safe, always refer to specific values of the type via symbolic expressions containing attributes like 'FIRST, 'LAST, 'PRED, and 'SUCC rather than via literal values.

The best solution is to introduce a new generic formal type parameter and use it in place of the subtype, as shown below:

```
-------------------------------------------------------------------------
generic
   type OBJECTS is range ;
   OBJECT : in out OBJECTS;
   with procedure PUT (PARAMETER : in OBJECTS);
   with function  GET return OBJECTS;
package INPUT_OUTPUT is
   ...
end INPUT_OUTPUT;
-------------------------------------------------------------------------
```

This is a clear statement by the developer of the generic unit that no assumptions are made about the OBJECTS type other than that it is an integer type. This should reduce the likelihood of any invalid assumptions being made in the body of the generic unit.

8.2.5 Overloading in Generic Units

guideline

- Be careful about overloading the names of subprograms exported by the same generic package.

example

```
-------------------------------------------------------------------------
generic
   type ITEMS is limited private;
package INPUT_OUTPUT is
   procedure PUT (ITEM : in INTEGER);
   procedure PUT (ITEM : in ITEMS);
end INPUT_OUTPUT;
-------------------------------------------------------------------------
```

rationale

If the generic package shown in the example above is instantiated with INTEGER (or any subtype of INTEGER) as the actual type corresponding to generic formal ITEM, then the two PUT procedures have identical interfaces, and all calls to PUT are ambiguous. Therefore, this package cannot be used with type INTEGER. In such a case, it is better to give unambiguous names to all subprograms. See section 12.3(22) of the Ada Language Reference Manual (Department of Defense 1983) for more information.

8.2.6 Hidden Tasks

guideline

- Document which generic formal parameters are accessed from a task hidden inside the generic unit.

rationale

Concurrent access to data structures must be carefully planned to avoid errors, especially for data structures which are not atomic (see Chapter 6 for details). If a generic unit accesses one of its generic formal parameters (reads or writes the value of a generic formal object or calls a generic formal subprogram which reads or writes data) from within a task contained in the generic unit, then there is the possibility of concurrent access for which the user may not have planned. In such a case, the user should be warned by a comment in the generic specification.

8.2.7 Exceptions

guideline

- Propagate exceptions out of reusable parts. Handle exceptions within reusable parts only when you are certain that the handling is appropriate in all circumstances.
- Always propagate exceptions raised by generic formal subprograms, after performing any cleanup necessary to the correct operation of future invocations of the generic instantiation.
- Always leave state variables unmodified when raising an exception.
- Always leave parameters unmodified when raising an exception.

example

```
-------------------------------------------------------------------------
generic
   type NUMBERS is limited private;
   with procedure GET (NUMBER : out NUMBERS);
procedure PROCESS_NUMBERS;
-------------------------------------------------------------------------
procedure PROCESS_NUMBERS is
   NUMBER : NUMBERS;
begin
   ...
   begin
      GET (NUMBER);
   exception
      when others =>
         PERFORM_CLEANUP_NECESSARY_FOR_PROCESS_NUMBERS;
         raise;
   end;
   ...
end PROCESS_NUMBERS;
-------------------------------------------------------------------------
```

rationale

On most occasions, an exception is raised because an undesired event (such as floating-point overflow) has occurred. Such events often need to be dealt with entirely differently with different uses of a particular software part. It is very difficult to anticipate all the ways that users of the part may wish to have the exceptions handled. Passing the exception out of the part is the safest treatment.

In particular, when an exception is raised by a generic formal subprogram, the generic unit is in no position to understand why or to know what corrective action to take. Therefore, such exceptions should always be propagated back to the caller of the generic instantiation. However, the generic unit must first clean up after itself, restoring its internal data structures to a correct state so that future calls may be made to it after the caller has dealt with the current exception. For this reason, all calls to generic formal subprograms should be within the scope of a `when others` exception handler if the internal state is modified, as shown in the example above.

When a reusable part is invoked, the user of the part should be able to know exactly what operation (at the appropriate level of abstraction) has been performed. For this to be possible, a reusable part must always do all or none of its specified function; it must never do half. Therefore, any reusable part which terminates early by raising or propagating an exception should return to the caller with no effect on the internal or external state. The easiest way to do this is to test for all possible exceptional conditions before making any state changes (modifying internal state variables, making calls to other reusable parts to modify their states, updating files, etc.). When this is not possible, it is best to restore all internal and external states to the values which were current when the part was invoked before raising or propagating the exception. When even this is not possible, it is important to document this poten' hazardous situation in the comment header of the specification of the part.

A similar problem arises with parameters of mode `out` or `in out` when exceptions are raised. The Ada language defines these modes in terms of "copy-in" and "copy-back" semantics, but leaves the actual parameter-passing mechanism undefined. When an exception is raised, the copy-back does not occur, but for an Ada compiler which passes parameters by reference, the actual parameter has already been updated. When parameters are passed by copy, the update does not occur. To reduce ambiguity, increase portability, and avoid situations where some but not all of the actual parameters are updated

when an exception is raised, it is best to treat values of out and in out parameters like state variables, updating them only after it is certain that no exception will be raised.

8.3 ADAPTABILITY

Reusable parts often need to be changed before they can be used in a specific application. They should therefore be structured so that change is easy and as localized as possible. One way of achieving adaptability is to create general parts with complete functionality, only a subset of which might be needed in a given application. Another is to use Ada generic construct to produce parts which can be appropriately instantiated with different parameters. Both of these approaches avoid the error-prone process of adapting a part by changing its code, but have limitations and can carry some overhead.

Anticipated changes, that is, changes that can be reasonably foreseen by the developer of the part, should be provided for as far as possible. Unanticipated change can only be accommodated by carefully structuring a part to be adaptable. Many of the considerations pertaining to maintainability apply. If the code is of high quality, clear, and conforms to well-established design principles such as information hiding, it is easier to adapt in unforeseen ways.

8.3.1 Complete Functionality

guideline

- Provide complete functionality in a reusable part or set of parts. Build in complete functionality, including end conditions, even if some functionality is not needed in this application.

example

```
INITIALIZE_QUEUE (...);              -- initialization operation
if STACK_FULL (...);                 -- probing operation
SYMBOL_TABLE.CLOSE_FRAME (...);      -- finalization operation
```

rationale

This is particularly important in designing/programming an abstraction. Completeness ensures that you have configured the abstraction correctly, without built-in assumptions about its execution environment. It also ensures the proper separation of functions so that they are useful to the current application and, in other combinations, to other applications. It is particularly important that they be available to other applications; remember that they can be "optimized" out of the final version of the current product.

note

The example illustrates end condition functions. An abstraction should be automatically initialized before its user gets a chance to damage it. When that is not possible, it should be supplied with initialization operations. In any case, it needs finalization operations, both explicit and default/automatic. Where possible, probing operations should be provided to determine when limits are about to be exceeded, so that the user can avoid causing exceptions to be raised.

It is also useful to provide reset operations for many objects. To see that a reset and an initiation can be different, consider the analogous situation of a "warm boot" and a "cold boot" on a personal computer.

Even if all of these operations are not appropriate for the abstraction, the exercise of considering them aids in formulating a complete set of operations, others of which may be used by another application.

Some implementations of the language link all subprograms of a package into the executable file, ignoring whether they are used or not, making unused operations a liability (see Guideline 8.4.4). In such cases, where the overhead is significant, create a copy of the fully functional part and comment out the unused operations with an indication that they are redundant in this application.

8.3.2 Generic Units

guideline

- Use generic units to avoid code duplication.
- Parameterize generic units for maximum adaptability.
- Reuse common instantiations of generic units, as well as the generic units themselves.

rationale

Ada does not allow subprograms or data types to be passed as actual parameters to subprograms during execution. Such parameters must be specified as generic formal parameters to a generic unit when it is instantiated. Therefore, if you want to write a subprogram for which there is variation from call to call in the data type of objects on which it operates, or in the subprogram which it calls, then you must write the subprogram as a generic unit and instantiate it once for each combination of data type and subprogram parameters. The instantations of the unit can then be called as regular subprograms.

If you find yourself writing two very similar routines differing only in the data type they operate on or the subprograms they call, then it is probably better to write the routine once as a generic unit and instantiate it twice to get the two versions you need. When the need arises later to modify the two routines, the change only needs to be made in one place. This greatly facilitates maintenance.

Once you have made such a choice, consider other aspects of the routine that these two instances may have in common but which are not essential to the nature of the routine. Factor these out as generic formal parameters. When the need arises later for a third similar routine, it can be automatically produced by a third instantiation, if you have foreseen all the differences between it and the other two. A parameterized generic unit can be very reusable.

It may seem that the effort involved in writing generic rather than nongeneric units is substantial. However, making units generic is not much more difficult or time-consuming than making them nongeneric once you become familiar with the generic facilities. It is, for the most part, a matter of practice. Also, any effort put into the development of the unit will be recouped when the unit is reused, as it surely will be if it is placed in a reuse library with sufficient visibility. Do not limit your thinking about potential reuse to the application you are working on or to other applications with which you are very familiar. Applications with which you are not familiar or future applications might be able to reuse your software.

After writing a generic unit and placing it in your reuse library, the first thing you are likely to do is to instantiate it once for your particular needs. At this time, it is a good idea to consider whether there are instantiations which are very likely to be widely used. If so, place each such instantiation in your reuse library so that they can be found and shared by others.

8.3.3 Using Generic Units to Encapsulate Algorithms

guideline

- Use generic units to encapsulate algorithms independently of data type.

example

This is the specification of a generic sort procedure:

```
-------------------------------------------------------------------------
generic
   type ELEMENT is limited private;
   type DATA    is array (POSITIVE range <>) of ELEMENT;
   with function  "<" (LEFT, RIGHT : in ELEMENT) return BOOLEAN is <>;
   with procedure SWAP (LEFT, RIGHT : in out ELEMENT) is <>;
procedure GENERIC_SORT (DATA_TO_SORT : in out DATA);
-------------------------------------------------------------------------
```

The generic body looks just like a regular procedure body and can make full use of the generic formal parameters in implementing the sort algorithm:

```
-------------------------------------------------------------------------
procedure GENERIC_SORT (DATA_TO_SORT : in out DATA) is
begin
   ...
   for I in DATA_TO_SORT'range loop
      ...
      if DATA_TO_SORT (I) < DATA_TO_SORT (J) then
         SWAP (DATA_TO_SORT (I), (DATA_TO_SORT (J));
      end if;
      ...
   end loop;
   ...
end GENERIC_SORT;
-------------------------------------------------------------------------
```

The generic procedure can be instantiated as:

```
type INTEGER_ARRAY is array (1..100) of INTEGER;
-------------------------------------------------------------------------
procedure SORT is
  new GENERIC_SORT (ELEMENT => INTEGER, DATA => INTEGER_ARRAY);
-------------------------------------------------------------------------
```

or

```
type STRING_80 is STRING (1..80);
type STRING_ARRAY is array (1..100) of STRING_80;
-------------------------------------------------------------------------
procedure SORT is
  new GENERIC_SORT (ELEMENT => STRING_80  DATA => STRING_ARRAY);
-------------------------------------------------------------------------
```

and called as:

```
INTEGER_ARRAY_1 : INTEGER_ARRAY;
...
SORT (INTEGER_ARRAY_1);
```

or

```
STRING_ARRAY_1  : STRING_ARRAY;
...
SORT (STRING_ARRAY_1);
```

rationale

A sort algorithm can be described independently of the data type being sorted. This generic procedure takes the ELEMENT data type as a generic limited private type parameter so that it assumes as little as possible about the data type of the objects actually being operated on. It also takes DATA as a generic formal parameter so that instantiations can have entire arrays passed to them for sorting. Finally, it explicitly requires the two operators that it needs to do the sort: comparison and swap. The sort algorithm is encapsulated without reference to any data type. The generic can be instantiated to sort an array of any data type.

8.3.4 Using Generic Units for Abstract Data Types

guideline

- Use abstract data types in preference to abstract data objects.
- Use generic units to implement abstract data types independently of their component data type.

example

This example presents a series of different techniques which can be used to generate abstract data types and objects. A discussion of the merits of each follows in the rationale section below. The first is an abstract data object (ADO), also known as an abstract state machine (ASM). It encapsulates one stack of integers.

```
-------------------------------------------------------------------------
package BOUNDED_STACK is
   subtype ELEMENTS is INTEGER;
   MAX_STACK_SIZE : constant := 100;
   procedure PUSH (ELEMENT : in     ELEMENTS);
   procedure POP  (ELEMENT :    out ELEMENTS);
   OVERFLOW  : exception;
   UNDERFLOW : exception;
   ...
end BOUNDED_STACK;
-------------------------------------------------------------------------
```

The second is an abstract data type (ADT). It differs from the ADO by exporting the STACKS type, which allows the user to declare any number of stacks of integers. Note that since multiple stacks may now exist, it is necessary to specify a STACK argument on calls to PUSH and POP.

```
-------------------------------------------------------------------------------
package BOUNDED_STACK is
   subtype ELEMENTS is INTEGER;
   type STACKS      is limited private;
   MAX_STACK_SIZE : constant := 100;
   procedure PUSH (STACK : in out STACKS; ELEMENT : in     ELEMENTS);
   procedure POP  (STACK : in out STACKS; ELEMENT :    out ELEMENTS);
   OVERFLOW  : exception;
   UNDERFLOW : exception;
   ...
private
   type STACK_INFO;
   type STACKS is access STACK_INFO;
end BOUNDED_STACK;
-------------------------------------------------------------------------------
```

The third is a parameterless generic abstract data object (GADO). It differs from the ADO (the first example) simply by being generic, so that the user can instantiate it multiple times to obtain multiple stacks of integers.

```
-------------------------------------------------------------------------------
generic
package BOUNDED_STACK is
   subtype ELEMENTS is INTEGER;
   MAX_STACK_SIZE : constant := 100;
   procedure PUSH (ELEMENT : in     ELEMENTS);
   procedure POP  (ELEMENT :    out ELEMENTS);
   OVERFLOW  : exception;
   UNDERFLOW : exception;
   ...
end BOUNDED_STACK;
-------------------------------------------------------------------------------
```

The fourth is a slight variant on the third, still a generic abstract data object (GADO) but with parameters. It differs from the third example by making the data type of the stack a generic parameter so that stacks of data types other than INTEGER can be created. Also, MAX_STACK_SIZE has been made a generic parameter which defaults to 100 but can be specified by the user, rather than a constant defined by the package.

```
-------------------------------------------------------------------------------
generic
   type ELEMENTS is limited private;
   with procedure ASSIGN (FROM : in ELEMENTS; TO : out ELEMENTS);
   MAX_STACK_SIZE : in NATURAL := 100;
package BOUNDED_STACK is
   procedure PUSH (ELEMENT : in     ELEMENTS);
   procedure POP  (ELEMENT :    out ELEMENTS);
   OVERFLOW  : exception;
   UNDERFLOW : exception;
   ...
end BOUNDED_STACK;
-------------------------------------------------------------------------------
```

Finally, the fifth is a generic abstract data type (GADT). It differs from the GADO in the fourth example in the same way that the ADT in the second example differed from the ADO in the first example; it exports the STACKS type, which allows the user to declare any number of stacks.

```
-------------------------------------------------------------------------------
generic
   type ELEMENTS is limited private;
   with procedure ASSIGN (FROM : in ELEMENTS; TO : out ELEMENTS);
   MAX_STACK_SIZE : in NATURAL := 100;
package BOUNDED_STACK is
   type STACKS is limited private;
   procedure PUSH (STACK : in out STACKS; ELEMENT : in     ELEMENTS);
   procedure POP  (STACK : in out STACKS; ELEMENT :    out ELEMENTS);
   OVERFLOW  : exception;
   UNDERFLOW : exception;
   ...
private
   type STACK_INFO;
   type STACKS is access STACK_INFO;
end BOUNDED_STACK;
-------------------------------------------------------------------------------
```

rationale

The biggest advantage of an ADT over an ADO (or a GADT over a GADO) is that the user of the package can declare as many objects as desired with an ADT. These objects can be declared as standalone variables or as components of arrays and records. They can also be passed as parameters. None of this is possible with an ADO, where the single data object is encapsulated inside of the package. Furthermore, an ADO provides no more protection of the data structure than an ADT. When a private type is exported by the ADT package, as in the example above, then for both the ADO and ADT, the only legal operations which can modify the data are those defined explicitly by the package (in this case, PUSH and POP). For these reasons, an ADT or GADT is almost always preferable to an ADO or GADO, respectively.

A GADO is similar to an ADT in one way: it allows multiple objects to be created by the user. With an ADT, multiple objects can be declared using the type defined by the ADT package. With a GADO (even a GADO with no generic formal parameters, as shown in the third example), the package can be instantiated multiple times to produce multiple objects. However, the similarity ends there. The multiple objects produced by the instantiations suffer from all restrictions described above for ADOs; they cannot be used in arrays or records or passed as parameters. Furthermore, the objects are each of a different type, and no operations are defined to operate on more than one of them at a time. For example, there cannot be an operation to compare two such objects or to assign one to another. The multiple objects declared using the type defined by an ADT package suffer from no such restrictions; they can be used in arrays and records and can be passed as parameters. Also, they are all declared to be of the same type, so that it is possible for the ADT package to provide operations to assign, compare, copy, etc. For these reasons, an ADT is almost always preferable to a parameterless GADO.

The biggest advantage of a GADT or GADO over an ADT or ADO, respectively, is that the GADT and GADO are generic and can thus be parameterized with types, subprograms, and other configuration information. Thus, as shown above, a single generic package can support bounded stacks of any data type and any stack size, while the ADT and ADO above are restricted to stacks of INTEGER, no more than 100 in size. For this reason, a GADO or GADT is almost always preferable to an ADO or ADT.

The list of examples above is given in order of increasing power and flexibility, starting with an ADO and ending with a GADT. These advantages are not expensive in terms of complexity or development time. The specification of the GADT above is not significantly harder to write or understand than the specification of the ADO. The bodies are also nearly identical. Compare the body for the simplest version, the ADO:

```
-------------------------------------------------------------------------
package body BOUNDED_STACK is
   type STACK_SLOTS is array (NATURAL range <>) of ELEMENTS;
   type STACK_INFO is
      record
         SLOTS  : STACK_SLOTS (1 .. MAX_STACK_SIZE);
         INDEX  : NATURAL := 0;
      end record;
   STACK : STACK_INFO;
   -------------------------------------------------------------------------
   procedure PUSH (ELEMENT : in ELEMENTS) is
   begin
      if STACK.INDEX >= MAX_STACK_SIZE then
         raise OVERFLOW;
      end if;
      STACK.INDEX := STACK.INDEX + 1;
      STACK.SLOTS (STACK.INDEX) := ELEMENT;
   end PUSH;
   -------------------------------------------------------------------------
   procedure POP (ELEMENT : out ELEMENTS) is
   begin
      if STACK.INDEX <= 0 then
         raise UNDERFLOW;
      end if;
      ELEMENT := STACK.SLOTS (STACK.INDEX);
      STACK.INDEX := STACK.INDEX - 1;
   end POP;
   -------------------------------------------------------------------------
   ...
end BOUNDED_STACK;
-------------------------------------------------------------------------
```

with the body for the most powerful and flexible version, the GADT:

```
-------------------------------------------------------------------------
package body BOUNDED_STACK is
   type STACK_SLOTS is array (NATURAL range <>) of ELEMENTS;
   type STACK_INFO is
      record
         SLOTS  : STACK_SLOTS (1 .. MAX_STACK_SIZE);
         INDEX : NATURAL := 0;
      end record;
   -------------------------------------------------------------------------
   procedure PUSH (STACK : in out STACKS; ELEMENT : in ELEMENTS) is
   begin
      if STACK.INDEX >= MAX_STACK_SIZE then
         raise OVERFLOW;
      end if;
      STACK.INDEX := STACK.INDEX + 1;
      ASSIGN (FROM => ELEMENT, TO => STACK.SLOTS (STACK.INDEX));
   end PUSH;
   -------------------------------------------------------------------------
   procedure POP (STACK : in out STACKS; ELEMENT : out ELEMENTS) is
   begin
      if STACK.INDEX <= 0 then
         raise UNDERFLOW;
      end if;
      ASSIGN (FROM => STACK.SLOTS (STACK.INDEX), TO => ELEMENT);
      STACK.INDEX := STACK.INDEX - 1;
   end POP;
   -------------------------------------------------------------------------
   ...
end BOUNDED_STACK;
-------------------------------------------------------------------------
```

There are only two differences. First, the ADO declares a local object called STACK, while the GADT has one additional parameter (called STACK) on each of the exported procedures PUSH and POP. Second, the GADT uses the ASSIGN procedure rather than the assignment operator ":=" because the generic formal type ELEMENT was declared `limited private`. This second difference could have been avoided by declaring ELEMENT as `private`, but this is not recommended because it reduces the composability of the generic reusable part.

8.3.5 Iterators

guideline

- Provide iterators for traversing complex data structures within reusable parts.
- Provide both active and passive iterators.
- Protect the iterators from errors due to modification of the data structure during iteration.
- Document the behavior of the iterators when the data structure is modified during traversal.

example

The following package defines an abstract list data type, with both active and passive iterators for traversing a list.

```
-------------------------------------------------------------------------
generic
   type ELEMENTS is limited private;
   with procedure ASSIGN (FROM : in ELEMENTS; TO : out ELEMENTS);
package UNBOUNDED_LIST is
   type LISTS is limited private;
   procedure INSERT (...);
   procedure REMOVE (...);

   -- Passive (generic) iterator.
   generic
      with procedure PROCESS (ELEMENT  : in out ELEMENTS;
                              CONTINUE :    out BOOLEAN);
   procedure ITERATE (LIST : in LISTS);

   -- Active iterator
   type ITERATORS is limited private;
   procedure INITIALIZE (ITERATOR : in out ITERATORS;
                         LIST     : in     LISTS);
   function  MORE       (ITERATOR : in     ITERATORS) return BOOLEAN;
   procedure ADVANCE    (ITERATOR : in out ITERATORS);
   function  CURRENT    (ITERATOR : in     ITERATORS) return ELEMENTS;
   procedure TERMINATE  (ITERATOR : in out ITERATORS);

   ...
private
   ...
end UNBOUNDED_LIST;
-------------------------------------------------------------------------
```

After instantiating the generic package, and declaring a list, as:

```
-------------------------------------------------------------------------
with UNBOUNDED_LIST;
procedure LIST_USER is
   type EMPLOYEES is ...;
   ---------------------------------------------------------------------
   procedure ASSIGN (FROM : in EMPLOYEES; TO : out EMPLOYEES);
   ---------------------------------------------------------------------
   package MY_LIST is
     new UNBOUNDED_LIST (ELEMENTS => EMPLOYEES, ASSIGN => ASSIGN);
   ---------------------------------------------------------------------
   EMPLOYEE_LIST : MY_LIST.LISTS;
```

the passive iterator is instantiated, specifying the name of the routine which should be called for each list element when the iterator is called.

```
   ---------------------------------------------------------------------
   procedure PROCESS_EMPLOYEE (EMPLOYEE : in out EMPLOYEES;
                               CONTINUE :    out BOOLEAN) is
   begin -- PROCESS_EMPLOYEE
      ...          -- Perform the required action for EMPLOYEE here.
      CONTINUE := TRUE;
   end PROCESS_EMPLOYEE;
   ---------------------------------------------------------------------
   procedure PROCESS_ALL_EMPLOYEES is
     new MY_LIST.ITERATE (PROCESS => PROCESS_EMPLOYEE);
   ---------------------------------------------------------------------
```

The passive iterator can then be called, as:

```
   ---------------------------------------------------------------------
begin -- LIST_USER
   PROCESS_ALL_EMPLOYEES (EMPLOYEE_LIST);
end LIST_USER;
-------------------------------------------------------------------------
```

Alternatively, the active iterator can be used, without the second instantiation required by the passive iterator, as:

```
   ITERATOR : MY_LIST.ITERATORS;
   EMPLOYEE : EMPLOYEES;
   ------------------------------------------------------------------------
begin -- LIST_USER
   MY_LIST.INITIALIZE (ITERATOR => ITERATOR, LIST => EMPLOYEE_LIST);
   while MY_LIST.MORE (ITERATOR) loop
      ASSIGN (FROM => MY_LIST.CURRENT (ITERATOR), TO => EMPLOYEE);
      MY_LIST.ADVANCE (ITERATOR);
      ...         -- Perform the required action for EMPLOYEE here.
   end loop;
   MY_LIST.TERMINATE (ITERATOR);
end LIST_USER;
------------------------------------------------------------------------
```

rationale

Iteration over complex data structures is often required and, if not provided by the part itself, can be difficult to implement without violating information hiding principles.

Active and passive iterators each have their advantages, but neither is appropriate in all situations. Therefore, it is recommended that both be provided to give the user a choice of which to use in each situation.

Passive iterators are simpler and less error-prone than active iterators, in the same way that the `for` loop is simpler and less error-prone than the `while` loop. There are fewer mistakes that the user can make in using a passive iterator. Simply instantiate it with the routine to be executed for each list element, and call the instantiation for the desired list. Active iterators require more care by the user. The iterator must be declared, then initialized with the desired list, then CURRENT and ADVANCE must be called in a loop until MORE returns false, then the iterator must be terminated. Care must be taken to perform these steps in the proper sequence. Care must also be taken to associate the proper iterator variable with the proper list variable. It is possible for a change made to the software during maintenance to introduce an error, perhaps an infinite loop.

On the other hand, active iterators are more flexible than passive iterators. With a passive iterator, it is difficult to perform multiple, concurrent, synchronized iterations. For example, it is much easier to use active iterators to iterate over two sorted lists, merging them into a third sorted list. Also, for multidimensional data structures, a small number of active iterator routines may be able to replace a large number of passive iterators, each of which implements one combination of the active iterators. Consider, for example, a binary tree. In what order should the passive iterator visit the nodes? Depth first? Breadth first? What about the need to do a binary search of the tree? Each of these could be implemented as a passive iterator, but it may make more sense to simply define the MORE_LEFT, MORE_RIGHT, ADVANCE_LEFT, and ADVANCE_RIGHT routines required by the active iterator to cover all combinations. Finally, active iterators can be passed as generic formal parameters while passive iterators cannot because passive iterators are themselves generic, and generic units cannot be passed as parameters to other generic units.

For either type of iterator, semantic questions can arise about what happens when the data structure is modified as it is being iterated. When writing an iterator, be sure to consider this possibility, and indicate with comments the behavior which occurs in such a case. It is not always obvious to the user what to expect. For example, to determine the "closure" of a mathematical "set" with respect to some operation, a common algorithm is to iterate over the members of the set, generating new elements and adding them to the set. In such a case, it is important that elements added to the set during the iteration be encountered subsequently during the iteration. On the other hand, for other algorithms it may be important that the set which it iterated is the set as it existed at the beginning of the iteration. In the case of a prioritized list data structure, if the list is iterated in priority order, it may be important that elements inserted at lower priority than the current element during iteration not be encountered subsequently during the iteration, but that elements inserted at a higher priority should be encountered. In any case, make a conscious decision about how the iterator should operate, and document that behavior in the package specification.

Deletions from the data structure also pose a problem for iterators. It is a common mistake for a user to iterate over a data structure, deleting it piece by piece during the iteration. If the iterator is not prepared for such a situation, it is possible to end up dereferencing a null pointer or committing a similar error. Such situations can be prevented by storing extra information with each data structure which indicates whether it is currently being iterated, and using this information to disallow any modifications to the data structure during iteration. When the data structure is declared as a `limited private` type, as should

usually be the case when iterators are involved, the only operations defined on the type are declared explicitly in the package which declares the type, making it possible to add such tests to all modification operations.

note

For further discussion of passive and active iterators, (see Ross 1989 and Booch 1987).

8.3.6 Private and Limited Private Types

guideline

- Use limited private (not private) for generic formal types, explicitly importing assignment and equality operations if required.
- Export limited private, private, or nonprivate types, as appropriate, from generic packages.
- Use mode `in out` rather than `out` for parameters of a generic formal subprogram, when the parameters are of an imported limited type.

example

The first example violates the guideline by having private (nonlimited) generic formal types.

```
-------------------------------------------------------------------------
generic
   type ITEMS is private;
   type KEYS  is private;
   with function KEY_OF  (LEFT : in ITEMS) return KEYS;
package LIST_MANAGER is
   type LISTS is limited private;
   ----------------------------------------------------------------------
   procedure INSERT   (LIST : in     LISTS;
                       ITEM : in     ITEMS);
   ----------------------------------------------------------------------
   procedure RETRIEVE (LIST : in     LISTS;
                       KEY  : in     KEYS;
                       ITEM : in out ITEMS);
   ----------------------------------------------------------------------
private
   ...
end LIST_MANAGER;
-------------------------------------------------------------------------
```

The second example is improved by using limited private generic formal types and importing the assignment operation for ITEMS and the equality operator for KEYS.

```
-------------------------------------------------------------------------
generic
   type ITEMS is limited private;
   type KEYS  is limited private;
   with procedure ASSIGN (FROM : in ITEMS; TO : in out ITEMS);
   with function "="     (LEFT, RIGHT : in KEYS) return BOOLEAN;
   with function KEY_OF  (LEFT : in ITEMS) return KEYS;
package LIST_MANAGER is
   type LISTS is limited private;
   ----------------------------------------------------------------------
   procedure INSERT   (LIST : in     LISTS;
                       ITEM : in     ITEMS);
   ----------------------------------------------------------------------
   procedure RETRIEVE (LIST : in     LISTS;
                       KEY  : in     KEYS;
                       ITEM : in out ITEMS);
   ----------------------------------------------------------------------
private
   ...
end LIST_MANAGER;
-------------------------------------------------------------------------
```

rationale

For a generic component to be usable in as many contexts as possible, it should minimize the assumptions that it makes about its environment and should make explicit any assumptions that are necessary. In Ada, the assumptions made by generic units can be stated explicitly by the types of the

generic formal parameters. A limited private generic formal type prevents the generic unit from making any assumptions about the structure of objects of the type or about operations defined for such objects. A private (nonlimited) generic formal type allows the assumption that assignment and equality comparison operations are defined for the type. Thus, a limited private data type cannot be specified as the actual parameter for a private generic formal type.

Therefore, generic formal types should almost always be limited private rather than just private. This restricts the operations available on the imported type within the generic unit body but provides maximum flexibility for the user of the generic unit. Any operations required by the generic body should be explicitly imported as generic formal subprograms. In the second example above, only the operations required for managing a list of items with keys are imported: ASSIGN provides the ability to store items in the list, and KEY_OF and "=" support determination and comparison of keys during retrieval operations. No other operations are required to manage the list. Specifically, there is no need to be able to assign keys or compare entire items for equality. Those operations would have been implicitly available if a private type had been used for the generic formal type, and any actual type for which they were not defined could not have been used with this generic unit.

The situation is reversed for types exported by a reusable part. For exported types, the restrictions specified by limited and limited private are restrictions on the user of the part, not on the part itself. To provide maximum capability to the user of a reusable part, export types with as few restrictions as possible. Apply restrictions as necessary only to protect the integrity of the exported data structures and the abstraction.

In the example above, the LISTS type is exported as limited private to hide the details of the list implementation and protect the structure of a list. Limited private is chosen over private to prevent the user from being able to use the predefined assignment operation. This is important if the list is implemented as an access type pointing to a linked lists of records, because the predefined assignment would make copies of the pointer, not copies of the entire list, which the user may not realize. If it is expected that the user needs the ability to copy lists, then a copy operation should be explicitly exported.

Because they are so restrictive, limited private types are not always the best choice for types exported by a reusable part. In a case where it makes sense to allow the user to make copies of and compare data objects, and when the underlying data type does not involve access types (so that the entire data structure gets copied or compared), then it is better to export a (nonlimited) private type. In cases where it does not detract from the abstraction to reveal even more about the type, then a nonprivate type (e.g., a numeric, enumerated, record, or array type) should be used.

For cases where limited private types are exported, the package should explicitly provide equality and assignment operations, if appropriate to the abstraction. Limited private is almost always appropriate for types implemented as access types. In such cases, predefined equality is seldom the most desirable semantics. In such cases, also consider providing both forms of assignment (assignment of a reference and assignment of a copy).

When the parameters are of an imported limited type, using mode `in out` instead of `out` for parameters of a generic formal subprogram is important for the following reason. Ada allows an `out` mode parameter of a limited private type on a subprogram only when the subprogram is declared in the visible part of the package which that declares the private type. See section 7.4.4(4) of the Ada Language Reference Manual (Department of Defense 1983). There is no such restriction in parameters of mode `in out`. The result of this is that if you define a generic with a limited generic formal type and a generic formal subprogram with an `out` parameter of that type, then the generic can only be instantiated with a limited private actual type if the package which declares that type also declares a subprogram with exactly the same profile (number and types or arguments and return value) as your generic formal subprogram. A potential user who wants to instantiate your generic with a limited type defined in another package will not be able to write a subprogram to pass as the generic actual.

note

It is possible (but clumsy) to redefine equality for nonlimited types. However, if a generic imports a (nonlimited) private type and uses equality, it will automatically use the predefined equality and not the user-supplied redefinition. This is another argument for using limited private generic formal parameters.

8.4 INDEPENDENCE

A reusable part should be as independent as possible of other reusable parts. A potential user is less inclined to reuse a part if that part requires the use of other parts which seem unnecessary. The "extra baggage" of the other parts wastes time and space. A user would like to be able to reuse only that part which is perceived as useful.

Note that the concept of a "part" is intentionally vague here. A single package does not need to be independent of each other package in a reuse library, if the "parts" from that library which are typically reused are entire subsystems. If the entire subsystem is perceived as providing a useful function, the entire subsystem is reused. However, the subsystem should not be tightly coupled to all the other subsystems in the reuse library, so that it is difficult or impossible to reuse the subsystem without reusing the entire library. Coupling between reusable parts should only occur when it provides a strong benefit perceptible to the user.

8.4.1 Using Generic Parameters to Reduce Coupling

guideline

- Minimize `with` clauses on reusable parts, especially on their specifications.
- Use generic parameters instead of `with` statements to reduce the number of context clauses on a reusable part.
- Use generic parameters instead of `with` statements to import portions of a package rather than the entire package.

example

A procedure like the following:

```
-------------------------------------------------------------------------
with PACKAGE_A;
procedure PRODUCE_AND_STORE_A (...) is
   ...
begin
   ...
   PACKAGE_A.PRODUCE (...);
   ...
   PACKAGE_A.STORE (...);
   ...
end PRODUCE_AND_STORE_A;
-------------------------------------------------------------------------
```

can be rewritten as a generic unit:

```
-------------------------------------------------------------------------
generic
   with procedure PRODUCE (...);
   with procedure STORE   (...);
procedure PRODUCE_AND_STORE;
-------------------------------------------------------------------------
procedure PRODUCE_AND_STORE is
   ...
begin
   ...
   PRODUCE (...);
   ...
   STORE (...);
   ...
end PRODUCE_AND_STORE;
-------------------------------------------------------------------------
```

and then instantiated:

```
-------------------------------------------------------------------------
with PACKAGE_A;
with PRODUCE_AND_STORE;
procedure PRODUCE_AND_STORE_A is
  new PRODUCE_AND_STORE
    (PRODUCE => PACKAGE_A.PRODUCE,
     (STORE  => PACKAGE_A.STORE);
-------------------------------------------------------------------------
```

rationale

Context (with) clauses specify the names of other units upon which this unit depends. Such dependencies cannot and should not be entirely avoided, but it is a good idea to minimize the number of them which occur in the specification of a unit. Try to move them to the body, leaving the specification independent of other units so that it is easier to understand in isolation. Also, organize your reusable parts in such a way that the bodies of the units do not contain large numbers of dependencies on each other. Partitioning your library into independent functional areas with no dependencies spanning the boundaries of the areas is a good way to start. Finally, reduce dependencies by using generic formal parameters instead of with statements, as shown in the example above. If the units in a library are too tightly coupled, then no single part can be reused without reusing most or all of the library.

The first (nongeneric) version of PRODUCE_AND_STORE_A above is difficult to reuse because it depends on PACKAGE_A which may not be general purpose or generally available. If the operation PRODUCE_AND_STORE has reuse potential which is reduced by this dependency, a generic unit and an instantiation should be produced as shown above. Note that the with clause for PACKAGE_A has been moved from the PRODUCE_AND_STORE generic procedure which encapsulates the reusable algorithm to the PRODUCE_AND_STORE_A instantiation. Instead of naming the package which provides the required operations, the generic unit simply lists the required operations themselves. This increases the independence and reusability of the generic unit.

This use of generic formal parameters in place of with clauses also allows visibility at a finer granularity. The with clause on the nongeneric version of PRODUCE_AND_STORE_A makes all of the contents of PACKAGE_A visible to PRODUCE_AND_STORE_A, while the generic parameters on the generic version make only the PRODUCE and STORE operations available to the generic instantiation.

8.4.2 Coupling Due to Pragmas

guideline

- Avoid pragma ELABORATE in reusable parts.
- Avoid pragma PRIORITY in tasks hidden in reusable parts.

rationale

Pragma ELABORATE controls the order of elaboration of one unit with respect to another. This is another way of coupling units and should be avoided when possible in reusable parts, because it restricts the number of configurations in which the reusable parts can be combined.

Pragma PRIORITY controls the priority of a task relative to all other tasks in a particular system. It is inappropriate in a reusable part which does not know anything about the requirements and importance of other parts of the systems in which it is reused. Give careful consideration to a reusable part which claims that it can only be reused if its embedded task has the highest priority in the system. No two such parts can ever be used together.

8.4.3 Part Families

guideline

- Create families of generic or other parts with similar specifications.

example

The Booch parts (Booch 1987) are an example of the application of this guideline.

rationale

Different versions of similar parts (e.g., bounded versus unbounded stacks) may be needed for different applications or to change the properties of a given application. Often, the different behavior required by these versions cannot be obtained using generic parameters. Providing a family of parts with similar specifications makes it easy for the programmer to select the appropriate one for the current application or to substitute a different one if the needs of the application change.

note

A reusable part which is structured from subparts which are members of part families is particularly easy to tailor to the needs of a given application by substitution of family members.

8.4.4 Conditional Compilation

guideline

- Structure reusable code to take advantage of dead code removal by the compiler.

example

```
-------------------------------------------------------------------------
   separate (MATRIX_MATH)
procedure INVERT ( ... ) is
   type ALGORITHM is (GAUSSIAN, PIVOTING, CHOLESKI, TRI_DIAGONAL);
   WHICH_ALGORITHM : constant ALGORITHM := CHOLESKI;
begin -- INVERT
   case WHICH_ALGORITHM is
      when GAUSSIAN     => ... ;
      when PIVOTING     => ... ;
      when CHOLESKI     => ... ;
      when TRI_DIAGONAL => ... ;
   end case;
end INVERT;
-------------------------------------------------------------------------
```

rationale

Some compilers omit object code corresponding to parts of the program which they detect can never be executed. Constant expressions in conditional statements take advantage of this feature where it is available, providing a limited form of conditional compilation. When a part is reused in an implementation that does not support this form of conditional compilation, this practice produces a clean structure which is easy to adapt by deleting or commenting out redundant code where it creates an unacceptable overhead.

caution

Be aware of whether your implementation supports dead code removal, and be prepared to take other steps to eliminate the overhead of redundant code if necessary.

8.4.5 Table-Driven Programming

guideline

- Write table-driven reusable parts where possible and appropriate.

example

The epitome of table-driven reusable software is a parser generation system. A specification of the form of the input data and of its output, along with some specialization code, is converted to tables that are to be "walked" by pre-existing code using predetermined algorithms in the parser produced. Other forms of "application generators" work similarly.

rationale

Table-driven (sometimes known as data-driven) programs have behavior that depends on data `with`'ed at compile time or read from a file at run-time. In appropriate circumstances, table-driven programming provides a very powerful way of creating general-purpose, easily tailorable, reusable parts.

note

Consider whether differences in the behavior of a general-purpose part could be defined by some data structure at compile- or run-time and, if so, structure the part to be table-driven. The approach is most likely to be applicable when a part is designed for use in a particular application domain but needs to be specialized for use in a specific application within the domain. Take particular care in commenting the structure of the data needed to drive the part.

8.5 SUMMARY

understanding and clarity

- Select the least restrictive names possible for reusable parts and their identifiers.
- Reserve the best name for a generic instantiation, using the second best for the generic unit itself.

- Use names which indicate the behavioral characteristics of the reusable part, as well as its abstraction.
- Do not use any abbreviations in identifier or unit names.

robustness

- Use symbolic constants and constant expressions to allow multiple dependencies to be linked to a small number of symbols.
- Use unconstrained array types for array formal parameters and array return values.
- Make the size of local variables depend on actual parameter size where appropriate.
- Minimize the number of assumptions made by a unit.
- For assumptions which cannot be avoided, use types to automatically enforce conformance.
- For assumptions which cannot be automatically enforced by types, add explicit checks to the code.
- Document all assumptions.
- Beware of using subtypes as type marks when declaring generic formal objects of type `in out`.
- Beware of using subtypes as type marks when declaring parameters or return values of generic formal subprograms.
- Use symbolic expressions of attributes rather than literal values in reference to generic formal objects, and parameter and return values of generic formal subprograms.
- Be careful about overloading the names of subprograms exported by the same generic package.
- Document which generic formal parameters are accessed from a task hidden inside the generic unit.
- Propagate exceptions out of reusable parts. Handle exceptions within reusable parts only when you are certain that the handling is appropriate in all circumstances.
- Always propagate exceptions raised by generic formal subprograms, after performing any cleanup necessary to the correct operation of future invocations of the generic instantiation.
- Always leave state variables unmodified when raising an exception.
- Always leave parameters unmodified when raising an exception.

adaptability

- Provide complete functionality in a reusable part or set of parts. Build in complete functionality, including end conditions, even if some functionality is not needed in this application.
- Use generic units to avoid code duplication.
- Parameterize generic units for maximum adaptability.
- Reuse common instantiations of generic units, as well as the generic units themselves.
- Use generic units to encapsulate algorithms independently of data type.
- Use abstract data types in preference to abstract data objects.
- Use generic units to implement abstract data types independently of their component data type.
- Provide iterators for traversing complex data structures within reusable parts.
- Provide both active and passive iterators.
- Protect the iterators from errors due to modification of the data structure during iteration.
- Document the behavior of the iterators when the data structure is modified during traversal.
- Use limited private (not private) for generic formal types, explicitly importing assignment and equality operations if required.
- Export limited private, private, or nonprivate types, as appropriate, from generic packages.
- Use mode `in out` rather than `out` for parameters of a generic formal subprogram, when the parameters are of an imported limited type.
- Minimize `with` clauses on reusable parts, especially on their specifications.

- Use generic parameters instead of `with` statements to reduce the number of context clauses on a reusable part.
- Use generic parameters instead of `with` statements to import portions of a package rather than the entire package.
- Avoid pragma ELABORATE in reusable parts.
- Avoid pragma PRIORITY in tasks hidden in reusable parts.
- Create families of generic or other parts with similar specifications.
- Structure reusable code to take advantage of dead code removal by the compiler.
- Write table-driven reusable parts where possible and appropriate.

CHAPTER 9
Instantiation

A number of guidelines in this book are generic in nature. That is, they present a general principle of good Ada style, such as consistent indentation of source text, but do not prescribe a particular instantiation of that principle. In order to allow this book to function as a coding standard, you will need a particular instantiation.

This chapter lists all the guidelines requiring instantiation, and shows the instantiation adopted for the examples in this book. You might want to consider this instantiation as a coding standard. A code formatter can enforce many of these standards or change code to meet them as needed.

9.1 HORIZONTAL SPACING

guideline (2.1.1)

- Use consistent spacing around delimiters.
- Use the same spacing as you would in regular prose.

instantiation

Specifically, leave at least one blank space in the following places, as shown in the examples throughout this book. More spaces may be required for the vertical alignment recommended in subsequent guidelines.

- Before and after the following delimiters and binary operators:

```
+      -      *      /      &
<      =      >      /=     <=     >=
:=     =>     |      ..
:
<>
```

- Outside of the quotes for string (") and character (') literals, except where prohibited below.
- Outside, but not inside, of parentheses.
- After commas (,) and semicolons (;).

Do not leave any blank spaces in the following places, even if this conflicts with the above recommendation.

- After the plus (+) and minus (-) signs when used as unary operators.
- Inside of label delimiters (<< >>).
- Before and after the following:

```
**
```

- Between multiple consecutive opening or closing parentheses.
- Before commas (,) and semicolons (;).

9.2 INDENTATION

guideline (2.1.2)

- Indent and align nested control structures, continuation lines, and embedded units consistently.
- Distinguish between indentation for nested control structures and for continuation lines.
- Use spaces for indentation, not the tab character (Nissen and Wallis 1984, §2.2).

instantiation

Specifically, the following indentation conventions are recommended, as shown in the examples throughout this book. Note that the minimum indentation is described. More spaces may be required for the vertical alignment recommended in subsequent guidelines.

- Use the recommended paragraphing shown in the (Ada Reference Manual 1983).
- Use three spaces as the basic unit of indentation for nesting.
- Use two spaces as the basic unit of indentation for continuation lines.

A label is outdented three spaces. A continuation line is indented two spaces:

```
<<label>>                          | <long statement with line break>
   <statement>                     |   <trailing part of same statement>
```

The if statement and the plain loop:

```
if <condition> then                | <name>:
   <statements>                    |    loop
elsif <condition> then             |       <statements>
   <statements>                    |       exit when <condition>;
else                               |       <statements>
   <statements>                    |    end loop;
end if;                            |
```

Loops with the for and while iteration schemes:

```
<name>:                            | <name>:
   for <scheme> loop               |    while <condition> loop
      <statements>                 |       <statements>
   end loop;                       |    end loop;
```

The block and the case statement as recommended in the (Ada Reference Manual 1983):

```
<name>:                            | case <expression> is
   declare                         |    when <choice> =>
      <declarations>               |       <statements>
   begin                           |    when <choice> =>
      <statements>                 |       <statements>
   exception                       |    when others =>
      when <choice> =>             |       <statements>
         <statements>              | end case;
      when others =>               |
         <statements>              |
   end <name>;                     |
```

These case statements save space over the Ada Reference Manual recommendation and depend on very short statement lists, respectively. Whichever you choose, be consistent.

```
case <expression> is               | case <expression> is
when <choice> =>                   |    when <choice> => <statements>
     <statements>                  |                     <statements>
when <choice> =>                   |    when <choice> => <statements>
     <statements>                  |    when others   => <statements>
when others =>                     | end case;
     <statements>                  |
end case;                          |
```

The various forms of selective wait and the timed and conditional entry calls:

```
select                                  | select
   when <guard> =>                      |    <entry call>;
      <accept statement>                |    <statements>
      <statements>                      | or
or                                      |    delay <interval>;
      <accept statement>                |    <statements>
      <statements>                      | end select;
or                                      |
   when <guard> =>                      |
      delay <interval>;                 |
      <statements>                      |
or                                      | select
   when <guard> =>                      |    <enter call>;
      terminate;                        |    <statements>
else                                    | else
   <statements>                         |    <statements>
end select;                             | end select;
```

The accept statement and a subunit:

```
accept <specification> do               |    separate (<parent unit>)
   <statements>                         | <proper body>
end <name>;                             |
```

Body stubs of the program units:

```
procedure <specification> is            | package body <name> is
   separate;                            |    separate;
                                        |
function  <specification>               | task body <name> is
  return <type> is                      |    separate;
   separate;                            |
```

Proper bodies of program units:

```
procedure <specification> is            | package body <name> is
   <declarations>                       |    <declarations>
begin                                   | begin
   <statements>                         |    <statements>
exception                               | exception
   when <choice> =>                     |    when <choice> =>
      <statements>                      |       <statements>
end <name>;                             | end <name>;
                                        |
function  <specification>               | task body <name> is
  return  <type name> is                |    <declarations>
   <declarations>                       | begin
begin                                   |    <statements>
   <statements>                         | exception
exception                               |    when <choice> =>
   when <choice> =>                     |       <statements>
      <statements>                      | end <name>;
end <name>;                             |
```

Context clauses on compilation units are arranged as a table and are indented so as not to obscure the introductory line of the unit itself. Generic formal parameters do not obscure the unit itself. Function, package, and task specifications use standard indent:

```
   with <name>,                         | function  <specification>
        <name>,                         |   return <type>;
        <name>;                         |
   use  <name>,                         | package <name> is
        <name>,                         |    <declarations>
        <name>;                         | private
<compilation unit>                      |    <declarations>
                                        | end <name>;
                                        |
generic -- <kind of unit> <name>        | task type <name> is
   <formal parameters>                  |    entry <declaration>
<compilation unit>                      | end <name>;
```

Instantiations of generic units, and indentation of a record:

```
procedure <name> is
   new <generic name> <actuals>

function <name> is
   new <generic name> <actuals>

package <name> is
   new <generic name> <actuals>
```

```
type ... is
   record
      <component list>
      case <discriminant name> is
         when <choice> =>
            <component list>
         when <choice> =>
            <component list>
      end case;
   end record;
```

Indentation for record alignment:

```
for <name> use
   record <alignment clause>
      <component clause>
   end record;
```

9.3 MORE ON ALIGNMENT

guideline (2.1.5)

- Align parameter modes and parentheses vertically.

instantiation

Specifically it is recommended that you:

- Place one formal parameter specification per line.
- Vertically align parameter names, colons, the reserved word `in`, the reserved word `out`, and parameter types.
- Place the first parameter specification on the same line as the subprogram or entry name. If any of the parameter types are forced beyond the *line length* limit, place the first parameter specification on a new line indented as for continuation lines.

9.4 PAGINATION

guideline (2.1.7)

- Highlight the top of each package or task specification, the top of each program unit body, and the begin and end statements of each program unit.

instantiation

Specifically, it is recommended that you:

- Use a line of dashes, beginning at the same column as the current indentation.
- Use the shorter of the two dashed lines if they are adjacent.
- Omit the dashed line above the `begin`.
- When putting a dashed line at the top of a compilation unit, put it before, not after, the context clauses.

9.5 SOURCE CODE LINE LENGTH

guideline (2.1.9)

- Adhere to a maximum line length limit for source code (Nissen and Wallis 1984, §2.3).

instantiation

Specifically, it is recommended that you:

- Limit source code line lengths to a maximum of 78 characters.

9.6 NUMBERS

guideline (3.1.2)

- Represent numbers in a consistent fashion.
- Represent literals in a radix appropriate to the problem.
- Use underscores to separate digits the same way commas (or spaces for nondecimal bases) would be used in handwritten text.
- When using scientific notation, make the E consistently either upper or lower case.
- In an alternate base, represent the alphabetic characters in either all upper case, or all lower case.

instantiation

- Decimal and octal numbers are grouped by threes beginning counting on either side of the radix point.
- The E is always capitalized in scientific notation.
- Use upper case for the alphabetic characters representing digits in bases above 10.
- Hexadecimal numbers are grouped by fours beginning counting on either side of the radix point.

9.7 CAPITALIZATION

guideline (3.1.3)

- Make reserved words and other elements of the program visually distinct from each other.

instantiation

- Use lower case for all reserved words.
- Use upper case for all other identifiers.

9.8 FILE HEADERS

guideline (3.3.2)

- Put a file header on each source file.
- Place ownership, responsibility, and history information for the file in the file header.

instantiation

- Put a copyright notice in the file header.
- Put the author's name and department in the file header.
- Put a revision history in the file header, including a summary of each change, the date, and the name of the person making the change.

9.9 PROGRAM UNIT SPECIFICATION HEADER

guideline (3.3.3)

- Put a header on the specification of each program unit.
- Place information required by the user of the program unit in the specification header.
- Do not repeat information (except unit name) in the specification header which is present in the specification.
- Explain what the unit does, not how or why it does it.
- Describe the complete interface to the program unit, including any exceptions it can raise and any global effects it can have.
- Do not include information about how the unit fits into the enclosing software system.

- Describe the performance (time and space) characteristics of the unit.

instantiation

- Put the name of the program unit in the header.
- Briefly explain the purpose of the program unit.
- For packages, describe the effects of the visible subprograms on each other, and how they should be used together.
- List all exceptions which can be raised by the unit.
- List all global effects of the unit.
- List preconditions and postconditions of the unit.
- List hidden tasks activated by the unit.
- Do not list the names of parameters of a subprogram.
- Do not list the names of subprograms of a package.
- Do not list the names of all other units used by the unit.
- Do not list the names of all other units which use the unit.

9.10 PROGRAM UNIT BODY HEADER

guideline (3.3.4)

- Place information required by the maintainer of the program unit in the body header.
- Explain how and why the unit performs its function, not what the unit does.
- Do not repeat information (except unit name) in the header which is readily apparent from reading the code.
- Do not repeat information (except unit name) in the body header which is available in the specification header.

instantiation

- Put the name of the program unit in the header.
- Record portability issues in the header.
- Summarize complex algorithms in the header.
- Record reasons for significant or controversial implementation decisions.
- Record discarded implementation alternatives, along with the reason for discarding them.
- Record anticipated changes in the header, especially if some work has already been done to the code to make the changes easy to accomplish.

9.11 NAMED ASSOCIATION

guideline (5.2.2)

- Use named parameter association in calls of infrequently used subprograms or entries with many formal parameters.
- Use named association for constants, expressions, and literals in aggregates.
- Use named association when instantiating generics.
- Use named association for clarification when the actual parameter is any literal or expression.
- Use named association when supplying a nondefault value to an optional parameter.

instantiation

- Use named parameter association in calls of subprograms or entries called from less than five places in a single source file or with more than two formal parameters.

9.12 ORDER OF PARAMETER DECLARATIONS

guideline (5.2.5)

- Declare parameters in a consistent order (Honeywell 1986).

instantiation

- All `in` parameters without default values are declared before any `in out` parameter.
- All `in out` parameters are declared before any `out` parameters.
- All `out` parameters are declared before any parameters with default values.
- All parameters with default values are declared last.
- The order of parameters within these groups is derived from the needs of the application.

9.13 NESTING

guideline (5.6.1)

- Minimize the depth of nested expressions (Nissen and Wallis 1984).
- Minimize the depth of nested control structures (Nissen and Wallis 1984).
- Try simplification heuristics.

instantiation

- Do not nest expressions or control structures beyond a nesting level of five.

9.14 GLOBAL ASSUMPTIONS

guideline (7.1.1)

- Make considered assumptions about the support provided for the following on potential target platforms:
  - Number of bits available for type `INTEGER`.
  - Number of decimal digits of precision available for floating point types.
  - Number of bits available for fixed-point types.
  - Number of characters per line of source text.
  - Number of bits for `universal_integer` expressions.
  - Number of seconds for the range of `DURATION`.
  - Number of milliseconds for `DURATION'SMALL`.

instantiation

These are minimum values (or minimum precision in the case of `DURATION'SMALL`) that a project or application might assume that an implementation provides. There is no guarantee that a given implementation provides more than the minimum, so these would be treated by the project or application as maximum values also.

- 16 bits available for type `INTEGER`.
- 6 decimal digits of precision available for floating point types.
- 32 bits available for fixed-point types.
- 72 characters per line of source text.
- 16 bits for `universal_integer` expressions.
- `-86_400 .. 86_400` seconds (1 day) for the range of `DURATION`.
- 20 milliseconds for `DURATION'SMALL`.

CHAPTER 10
Complete Example

This chapter contains an example program to illustrate use of the guidelines. The program implements a simple menu-driven user interface that could be used as the front end for a variety of applications. It consists of a package for locally defined types (SPC_NUMERIC_TYPES), instantiations of I/O packages for those types (found in **spc_int_io_.a** and **spc_real_io_.a**), a package to perform ASCII terminal I/O for generating menus, writing prompts and receiving user input (TERMINAL_IO), and finally an example using the terminal I/O routines (EXAMPLE).

Within TERMINAL_IO, subprogram names are overloaded when several subprograms perform the same general function but for different data types.

The body for TERMINAL_IO uses separate compilation capabilities for a subprogram, DISPLAY_MENU, that is larger and more involved than the rest. Note that all literals that would be required are defined as constants. Nested loops, where they exist, are also named. The function defined in the file **terminal_io.a** on line 63 encapsulates a local exception handler within a loop. Where locally defined types could not be used, there is a comment explaining the reason. The use of short circuit control forms, both on an if statement and an exit are also illustrated.

The information that would have been in the file headers is redundant since it is contained in the title page of this book. The file headers are omitted from the following listings.

FILE: numerics_.a

```
 1:------------------------------------------------------------
 2:package SPC_NUMERIC_TYPES is
 3:
 4:   type TINY_INTEGER   is range -(2**7)  .. (2**7)  - 1;
 5:
 6:   type MEDIUM_INTEGER is range -(2**15) .. (2**15) - 1;
 7:
 8:   type BIG_INTEGER    is range -(2**31) .. (2**31) - 1;
 9:
10:   subtype TINY_NATURAL
11:     is TINY_INTEGER   range 0 .. TINY_INTEGER'LAST;
12:
13:   subtype MEDIUM_NATURAL
14:     is MEDIUM_INTEGER range 0 .. MEDIUM_INTEGER'LAST;
15:
16:   subtype BIG_NATURAL
17:     is BIG_INTEGER    range 0 .. BIG_INTEGER'LAST;
18:
19:   subtype TINY_POSITIVE
20:     is TINY_INTEGER   range 1 .. TINY_INTEGER'LAST;
21:
22:   subtype MEDIUM_POSITIVE
23:     is MEDIUM_INTEGER range 1 .. MEDIUM_INTEGER'LAST;
24:
25:   subtype BIG_POSITIVE
26:     is BIG_INTEGER    range 1 .. BIG_INTEGER'LAST;
27:
28:   type MEDIUM_FLOAT is digits 6;
29:   type BIG_FLOAT    is digits 9;
30:
31:   subtype PROBABILITIES is MEDIUM_FLOAT  range 0.0 .. 1.0;
32:
33:  ------------------------------------------------------------
34:   function MIN (LEFT  : in TINY_INTEGER;
35:                 RIGHT : in TINY_INTEGER)
36:     return TINY_INTEGER;
37:   ------------------------------------------------------------
38:   function MAX (LEFT  : in TINY_INTEGER;
39:                 RIGHT : in TINY_INTEGER)
40:     return TINY_INTEGER;
41:   ------------------------------------------------------------
42:   -- Additional function declarations
43:   -- to return the minimum and maximum values for each type.
44:end SPC_NUMERIC_TYPES;
45:------------------------------------------------------------
```

FILE: numerics.a

```
 1:-------------------------------------------------------------
 2:package body SPC_NUMERIC_TYPES is
 3:
 4:   -------------------------------------------------------------
 5:   function MIN (LEFT  : in TINY_INTEGER;
 6:                 RIGHT : in TINY_INTEGER)
 7:     return TINY_INTEGER is
 8:   begin -- MIN
 9:      if LEFT < RIGHT then
10:         return LEFT;
11:      else
12:         return RIGHT;
13:      end if;
14:   end MIN;
15:   -------------------------------------------------------------
16:   function MAX (LEFT  : in TINY_INTEGER;
17:                 RIGHT : in TINY_INTEGER)
18:     return TINY_INTEGER is
19:
20:   begin -- MAX
21:      if LEFT > RIGHT then
22:         return LEFT;
23:      else
24:         return RIGHT;
25:      end if;
26:   end MAX;
27:   -------------------------------------------------------------
28:   -- Additional functions to return minimum and maximum
29:   -- value for each type defined in the package.
30:   -------------------------------------------------------------
31:end SPC_NUMERIC_TYPES;
32:-------------------------------------------------------------
```

FILE: spc_int_io_.a

```
 1:-------------------------------------------------------------
 2:with SPC_NUMERIC_TYPES,
 3:     TEXT_IO;
 4:package SPC_SMALL_INTEGER_IO is new
 5:  TEXT_IO.INTEGER_IO (SPC_NUMERIC_TYPES.TINY_INTEGER);
 6:-------------------------------------------------------------
 7:with SPC_NUMERIC_TYPES,
 8:     TEXT_IO;
 9:package MEDIUM_INTEGER_IO is new
10:  TEXT_IO.INTEGER_IO (SPC_NUMERIC_TYPES.MEDIUM_INTEGER);
11:-------------------------------------------------------------
12:with SPC_NUMERIC_TYPES,
13:     TEXT_IO;
14:package BIG_INTEGER_IO is new
15:  TEXT_IO.INTEGER_IO (SPC_NUMERIC_TYPES.BIG_INTEGER);
16:-------------------------------------------------------------
```

FILE: spc_real_io_.a

```
 1:-------------------------------------------------------------
 2:with SPC_NUMERIC_TYPES,
 3:     TEXT_IO;
 4:package MEDIUM_FLOAT_IO is new
 5:  TEXT_IO.FLOAT_IO (SPC_NUMERIC_TYPES.MEDIUM_FLOAT);
 6:-------------------------------------------------------------
 7:with SPC_NUMERIC_TYPES,
 8:     TEXT_IO;
 9:package BIG_FLOAT_IO is new
10:  TEXT_IO.FLOAT_IO (SPC_NUMERIC_TYPES.BIG_FLOAT);
11:-------------------------------------------------------------
```

FILE: terminal_io_.a

```
 1:------------------------------------------------------------
 2:with SPC_NUMERIC_TYPES;
 3:use  SPC_NUMERIC_TYPES;
 4:
 5:package TERMINAL_IO is
 6:
 7:   MAX_FILE_NAME : constant := 30;
 8:   MAX_LINE      : constant := 30;
 9:
10:   subtype ALPHA_NUMERICS is CHARACTER range '0' .. 'Z';
11:   subtype LINES is STRING (1 .. MAX_LINE);
12:
13:   EMPTY_LINE : constant LINES := (others => ' ');
14:
15:   type MENUS is array (ALPHA_NUMERICS) of LINES;
16:
17:   subtype FILE_NAMES is STRING (1 .. MAX_FILE_NAME);
18:   ------------------------------------------------------------
19:   procedure GET_FILE_NAME (PROMPT      : in     STRING;
20:                           NAME        :    out FILE_NAMES;
21:                           NAME_LENGTH :    out NATURAL);
22:   ------------------------------------------------------------
23:   function YES (PROMPT : STRING) return BOOLEAN;
24:   ------------------------------------------------------------
25:   function GET (PROMPT : STRING) return MEDIUM_INTEGER;
26:   ------------------------------------------------------------
27:   function GET (PROMPT : STRING) return MEDIUM_FLOAT;
28:   ------------------------------------------------------------
29:   procedure DISPLAY_MENU (TITLE   : in     STRING;
30:                           OPTIONS : in     MENUS;
31:                           CHOICE  :    out ALPHA_NUMERICS);
32:   ------------------------------------------------------------
33:   procedure PAUSE (PROMPT : STRING);
34:   ------------------------------------------------------------
35:   procedure PAUSE;
36:   ------------------------------------------------------------
37:   procedure PUT (INTEGER_VALUE : MEDIUM_INTEGER);
38:   ------------------------------------------------------------
39:   procedure PUT (REAL_VALUE : MEDIUM_FLOAT);
40:   ------------------------------------------------------------
41:   procedure PUT (LABEL         : STRING;
42:                  INTEGER_VALUE : MEDIUM_INTEGER);
43:   ------------------------------------------------------------
44:   procedure PUT (LABEL      : STRING;
45:                  REAL_VALUE : MEDIUM_FLOAT);
46:   ------------------------------------------------------------
47:   procedure PUT_LINE (INTEGER_VALUE : MEDIUM_INTEGER);
48:   ------------------------------------------------------------
49:   procedure PUT_LINE (REAL_VALUE : MEDIUM_FLOAT);
50:   ------------------------------------------------------------
51:   procedure PUT_LINE (LABEL         : STRING;
52:                       INTEGER_VALUE : MEDIUM_INTEGER);
53:   ------------------------------------------------------------
54:   procedure PUT_LINE (LABEL      : STRING;
55:                       REAL_VALUE : MEDIUM_FLOAT);
56:   ------------------------------------------------------------
57:end TERMINAL_IO;
58:------------------------------------------------------------
```

FILE: terminal_io.a

```
 1:------------------------------------------------------------
 2:with MEDIUM_INTEGER_IO,
 3:     MEDIUM_FLOAT_IO,
 4:     TEXT_IO;
 5:
 6:use  TEXT_IO;
 7:
 8:package body TERMINAL_IO is  -- simple terminal i/o routines
 9:
10:   subtype RESPONSE is STRING (1 .. 20);
11:
```

```
12:   PROMPT_COLUMN    : constant           := 30;
13:   QUESTION_MARK    : constant STRING    := " ? ";
14:   STANDARD_PROMPT  : constant STRING    := " ==> ";
15:   BLANK            : constant CHARACTER := ' ';
16:   ------------------------------------------------------------
17:   procedure PUT_PROMPT (PROMPT   : STRING;
18:                         QUESTION : BOOLEAN := FALSE) is
19:   begin -- PUT_PROMPT
20:      PUT (PROMPT);
21:      if QUESTION then
22:         PUT (QUESTION_MARK);
23:      end if;
24:      SET_COL (PROMPT_COLUMN);
25:      PUT (STANDARD_PROMPT);
26:   end PUT_PROMPT;
27:   ------------------------------------------------------------
28:   function YES (PROMPT : STRING) return BOOLEAN is
29:
30:     RESPONSE_STRING        : RESPONSE := (others => BLANK);
31:     RESPONSE_STRING_LENGTH : NATURAL;
32:     ------------------------------------------------------------
33:   begin -- YES
34:      GET_RESPONSE:
35:      loop
36:         PUT_PROMPT (PROMPT, QUESTION => TRUE);
37:         GET_LINE (RESPONSE_STRING, RESPONSE_STRING_LENGTH);
38:         for POSITION in 1 .. RESPONSE_STRING_LENGTH loop
39:            if RESPONSE_STRING (POSITION) /= BLANK then
40:              return ((RESPONSE_STRING (POSITION) = 'Y') or
41:                      (RESPONSE_STRING (POSITION) = 'y'));
42:            end if;
43:         end loop;
44:         NEW_LINE; -- issue prompt until non-blank response
45:      end loop GET_RESPONSE;
46:   end YES;
47:   ------------------------------------------------------------
48:   procedure GET_FILE_NAME
49:     (PROMPT      : in     STRING;
50:      NAME        :    out FILE_NAMES;
51:      NAME_LENGTH :    out NATURAL) is
52:   begin -- GET_FILE_NAME
53:      PUT_PROMPT (PROMPT);
54:      GET_LINE (NAME, NAME_LENGTH);
55:   end GET_FILE_NAME;
56:   ------------------------------------------------------------
57:   function GET (PROMPT : STRING) return MEDIUM_INTEGER is
58:
59:      RESPONSE_STRING: RESPONSE := (others => BLANK);
60:      LAST           : NATURAL; -- Required by GET_LINE.
61:      VALUE          : MEDIUM_INTEGER;
62:     ------------------------------------------------------------
63:   begin -- GET
64:      loop
65:         begin
66:            PUT_PROMPT (PROMPT);
67:            GET_LINE (RESPONSE_STRING, LAST);
68:            VALUE:=
69:              MEDIUM_INTEGER'VALUE (RESPONSE_STRING (1 .. LAST));
70:            return VALUE;
71:         exception
72:            when others =>
73               PUT_LINE ("Please enter an integer");
74:         end;
75:      end loop;
76:   end GET;
77:   ------------------------------------------------------------
78:   procedure DISPLAY_MENU (TITLE   : in     STRING;
79:                           OPTIONS : in     MENUS;
80:                           CHOICE  :    out ALPHA_NUMERICS)
81:     is separate;
82:   ------------------------------------------------------------
83:   procedure PAUSE (PROMPT: STRING) is
84:   begin -- PAUSE
85:      PUT_LINE (PROMPT);
```

```
   PAUSE;
end PAUSE;
-------------------------------------------------------------
procedure PAUSE is
   BUFFER : RESPONSE;
   LAST   : NATURAL;
begin -- pause
   PUT ("Press return to continue");
   GET_LINE (BUFFER, LAST);
end PAUSE;
-------------------------------------------------------------
function GET (PROMPT : STRING) return MEDIUM_FLOAT is

   VALUE: MEDIUM_FLOAT;
   -----------------------------------------------------------
begin -- GET
   loop
      begin
         PUT_PROMPT (PROMPT);
         MEDIUM_FLOAT_IO.GET (VALUE);
         SKIP_LINE;
         return VALUE;
      exception
         when others =>
            SKIP_LINE;
            PUT_LINE ("Please enter a real number");
      end;
   end loop;
end GET;
-------------------------------------------------------------
procedure PUT (INTEGER_VALUE : MEDIUM_INTEGER) is
begin -- PUT
   MEDIUM_INTEGER_IO.PUT (INTEGER_VALUE, WIDTH => 4);
end PUT;
-------------------------------------------------------------
procedure PUT (REAL_VALUE : MEDIUM_FLOAT) is
begin -- PUT
   MEDIUM_FLOAT_IO.PUT (REAL_VALUE, FORE => 4,  AFT => 3,
     EXP => 0);
end PUT;
-------------------------------------------------------------
procedure PUT (LABEL         : STRING;
               INTEGER_VALUE : MEDIUM_INTEGER) is
begin -- PUT
   TEXT_IO.PUT (LABEL);
   MEDIUM_INTEGER_IO.PUT (INTEGER_VALUE);
end PUT;
-------------------------------------------------------------
procedure PUT (LABEL      : STRING;
               REAL_VALUE : MEDIUM_FLOAT) is
begin -- PUT
   TEXT_IO.PUT (LABEL);
   MEDIUM_FLOAT_IO.PUT (REAL_VALUE, FORE => 4, AFT => 3,
     EXP => 0);
end PUT;
-------------------------------------------------------------
procedure PUT_LINE (INTEGER_VALUE : MEDIUM_INTEGER) is
begin -- PUT_LINE
   TERMINAL_IO.PUT (INTEGER_VALUE);
   TEXT_IO.NEW_LINE;
end PUT_LINE;
-------------------------------------------------------------
procedure PUT_LINE (REAL_VALUE : MEDIUM_FLOAT) is
begin -- PUT_LINE
   TERMINAL_IO.PUT (REAL_VALUE);
   TEXT_IO.NEW_LINE;
end PUT_LINE;
-------------------------------------------------------------
procedure PUT_LINE (LABEL         : STRING;
                    INTEGER_VALUE : MEDIUM_INTEGER) is
begin -- PUT_LINE
   TERMINAL_IO.PUT (LABEL, INTEGER_VALUE);
   TEXT_IO.NEW_LINE;
end PUT_LINE;
```

```
160:   ------------------------------------------------------------
161:   procedure PUT_LINE (LABEL      : STRING;
162:                       REAL_VALUE : MEDIUM_FLOAT) is
163:   begin -- PUT_LINE
164:      TERMINAL_IO.PUT (LABEL, REAL_VALUE);
165:      TEXT_IO.NEW_LINE;
166:   end PUT_LINE;
167:   ------------------------------------------------------------
168:
169:end TERMINAL_IO;
170:------------------------------------------------------------
```

FILE: terminal_io__display_menu.a

```
 1:------------------------------------------------------------
 2:separate (TERMINAL_IO)
 3:procedure DISPLAY_MENU (TITLE   : in     STRING;
 4:                        OPTIONS : in     MENUS;
 5:                        CHOICE  :    out ALPHA_NUMERICS) is
 8:
 7:   LEFT_COLUMN  : constant         := 15;
 8:   RIGHT_COLUMN : constant         := 20;
 9:   PROMPT       : constant STRING := " ==> ";
10:
11:   type ALPHA_ARRAY is array (ALPHA_NUMERICS) of BOOLEAN;
12:
13:   VALID        : BOOLEAN;
14:   VALID_OPTION : ALPHA_ARRAY := (others => FALSE);
15:   ------------------------------------------------------------
16:   procedure DRAW_MENU (TITLE   : STRING;
17:                        OPTIONS : MENUS) is
18:   begin -- DRAW_MENU
19:      NEW_PAGE;
20:      NEW_LINE;
21:      SET_COL (RIGHT_COLUMN);
22:      PUT_LINE (TITLE);
23:      NEW_LINE;
24:      for CHOICE in ALPHA_NUMERICS loop
25:         if OPTIONS (CHOICE) /= EMPTY_LINE then
26:            VALID_OPTION (CHOICE) := TRUE;
27:            SET_COL (LEFT_COLUMN);
28:            PUT (CHOICE & " -- ");
29:            PUT_LINE (OPTIONS (CHOICE));
30:         end if;
31:      end loop;
32:   end DRAW_MENU;
33:   ------------------------------------------------------------
34:   procedure GET_RESPONSE (VALID  : out BOOLEAN;
35:                           CHOICE : out ALPHA_NUMERICS) is
36:
37:      BUFFER_SIZE : constant                := 20;
38:      DUMMY       : constant ALPHA_NUMERICS := 'X';
39:
40:      FIRST_CHAR  : CHARACTER;
41:      BUFFER      : STRING (1 .. BUFFER_SIZE);
42:
43:      -- IMPLEMENTATION NOTE:
44:      -- The following two declarations do not use
45:      -- locally defined types because a variable of type
46:      -- NATURAL is required by the TEXT_IO routines for
47:      -- strings, and there is no relational operator defined
48:      -- for our local TINY_, MEDIUM_  or BIG_POSITIVE and
49:      -- the standard type NATURAL.
50:      LAST        : NATURAL;
51:      INDEX       : POSITIVE;
52:      ------------------------------------------------------
53:      function UPPER_CASE (CURRENT_CHAR : CHARACTER)
54:        return CHARACTER is
55:
56:         CASE_DIFFERENCE : constant := 16#20#;
57:         ---------------------------------------------------
58:      begin -- UPPER_CASE
59:         if CURRENT_CHAR in 'a' .. 'z' then
60:            return CHARACTER'VAL (CHARACTER'POS (CURRENT_CHAR)
```

```
                                        - CASE_DIFFERENCE);
         else
            return CURRENT_CHAR;
         end if;
      end UPPER_CASE;
      ------------------------------------------------------
   begin -- GET_RESPONSE

      NEW_LINE;
      SET_COL (LEFT_COLUMN);
      PUT (PROMPT);

      GET_LINE (BUFFER, LAST);

      INDEX := POSITIVE'FIRST;
      loop
         exit when ((INDEX          >= LAST          ) or else
                    (BUFFER (INDEX) in ALPHA_NUMERICS)        );
         INDEX := POSITIVE'SUCC (INDEX);
      end loop;

      FIRST_CHAR := UPPER_CASE (BUFFER (INDEX));

      if (FIRST_CHAR not in ALPHA_NUMERICS) or else
         (not VALID_OPTION (FIRST_CHAR)   )        then
         VALID  := FALSE;
         CHOICE := DUMMY;
      else
         VALID  := TRUE;
         CHOICE := FIRST_CHAR;
      end if;

   end GET_RESPONSE;
   ------------------------------------------------------
   procedure BEEP is
   begin
      PUT (ASCII.BEL);
   end BEEP;
   ------------------------------------------------------
begin -- DISPLAY_MENU
   loop
      DRAW_MENU (TITLE, OPTIONS);
      GET_RESPONSE (VALID, CHOICE);
      exit when VALID;
      BEEP;
   end loop;
end DISPLAY_MENU;
------------------------------------------------------------
```

FILE: example.a

```
 1:------------------------------------------------------------------
 2:with SPC_NUMERIC_TYPES,
 3:     TERMINAL_IO;
 4:------------------------------------------------------------------
 5:procedure EXAMPLE is
 6:
 7:   package TIO renames TERMINAL_IO;
 8:
 9:   EXAMPLE_MENU : constant TIO.MENUS :=
10:      TIO.MENUS'('A'    => "Add item                    ",
11:                 'D'    => "Delete item                 ",
12:                 'M'    => "Modify item                 ",
13:                 'Q'    => "Quit                        ",
14:                 others => TIO.EMPTY_LINE);
15:
16:   USER_CHOICE : TIO.ALPHA_NUMERICS;
17:   ITEM        : SPC_NUMERIC_TYPES.MEDIUM_INTEGER;
18:   ------------------------------------------------------------
19:begin -- EXAMPLE
20:
21:   loop
22:      TIO.DISPLAY_MENU ("Example Menu", EXAMPLE_MENU,
23:        USER_CHOICE);
24:
25:      case USER_CHOICE is
26:         when 'A'    =>
27:            ITEM := TIO.GET ("Item to add");
28:         when 'D'    =>
29:            ITEM := TIO.GET ("Item to delete");
30:         when 'M'    =>
31:            ITEM := TIO.GET ("Item to modify");
32:         when 'Q'    =>
33:            exit;
34:         when others => -- error has already been
35:            null;       -- signaled to user
36:      end case;
37:
38:   end loop;
39:
40:end EXAMPLE;
41:------------------------------------------------------------------
42:
43:--  This is what is displayed, anything but A, D, M or Q beeps
44:--
45:--                  Example Menu
46:--
47:--            A -- Add item
48:--            D -- Delete item
49:--            M -- Modify item
50:--            Q -- Quit
51:--
52:--             ==>
```

APPENDIX A
Map from Ada Language Reference Manual to Guidelines

REFERENCES

Anderson, T. and R.W. Witty 1978 — Safe Programming. *BIT (Tidscrift Nordisk for Informations behandling)* 18:1-8.

ARTEWG 1986 — *Catalogue of Ada Runtime Implementation Dependencies*, draft version. Association for Computing Machinery, Special Interest Group for Ada, Ada Run-Time Environments Working Group.

Barnes, J.G.P. 1989 — *Programming in Ada*. third edition. Reading, MA.: Addison-Wesley.

Booch, G. 1987 — *Software Components with Ada - Structures, Tools and Subsystems*. Menlo Park, Ca.: The Benjamin/Cummings Publishing Company Inc.

Booch, G. 1987 — *Software Engineering with Ada*. second edition. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Charrette, R.N. 1986. — *Software Engineering Environments Concepts and Technology*. Intertext Publications Inc. New York: McGraw-Hill Inc.

Cohen, N.H. 1986 — *Ada as a Second Language*. New York: McGraw-Hill Inc.

Conti, R.A. 1987 — Critical Run-Time Design Tradeoffs in an Ada Implementation. *Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium*. pp. 486-495.

Department of Defense, Ada Joint Program Office. 1983. — *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A.

Foreman, J. and J. Goodenough 1987 — *Ada Adoption Handbook: A Program Manager's Guide*. Version 1.0, CMU/SEI-87-TR-9 ESD-TR-87-110. Software Engineering Institute.

MacLaren, L. 1980 — Evolving Toward Ada in Real Time Systems. *ACM Sigplan Notices*. 15(11):146–155.

Matthews, E.R. 1987 — Observations on the Portability of Ada I/O. *ACM Ada Letters*. VII(5):100–103.

Melliar-Smith, P.M. and B. Randell 1987 — Software Reliability: The Role of Programmed Exception Handling. *ACM Sigplan Notices*. 12(3):95–100.

Mowday, B.L. and E. Normand 1986 — *Ada Programming Standards*. General Dynamics Data Systems Division Departmental Instruction 414.717.

NASA 1987 — *Ada Style Guide*. Version 1.1, SEL–87–002. Goddard Space Flight Center: Greenbelt, MD 20771.

Nissen, J. and P. Wallis 1984 — *Portability and Style in Ada*. Cambridge University Press.

Pappas, F. 1985 — *Ada Portability Guidelines*. DTIC/NTIS #AD–A160 390.

Pyle, I.C. 1985 — *The Ada Programming Language*. second edition. UK.: Prentice–Hall International.

Rosen, J. P. 1987 — In Defense of the 'Use' Clause. *ACM Ada Letters*. VII(7):77–81.

Schneiderman, B. 1986 — Empirical Studies of Programmers: The Territory, Paths and Destinations. *Empirical Studies of Programmers*. ed. E. Soloway and S. Iyengar. pp. 1–12. Norwood, NJ: Ablex Publishing Corp.

Soloway, E., J. Pinto, S. Fertig, S. Letovsky, R. Lampert, D. Littman, and K. Ewing. 1986. — Studying Software Documentation From A Cognitive Perspective: A Status Report. *Proceedings of the Eleventh Annual Software Engineering Workshop*. Report SEL–86–006, Software Engineering Laboratory Greenbelt, Maryland:NASA Goddard Space Flight Center.

St.Dennis, R. 1986 — *A Guidebook for Writing Reusable Source Code in Ada* –Version 1.1. Report CSC–86–3:8213. Golden Valley, Minnesota: Honeywell Corporate Systems Development Division.

United Technologies 1987 — *CENC Programmer's Guide*. Appendix A Ada Programming Standards.

Volz, R.A., Mudge, Naylor and Mayer. 1985 — Some Problems in Distributing Real-time Ada Programs Across Machines. *Ada in Use, Proceedings of the Ada International Conference*. pp. 14–16. Paris.

BIBLIOGRAPHY

ACVC (Ada Compiler Validation Capability). Ada Validation Facility, ASD/SIOL. Wright-Patterson Air Force Base, OH.

Anderson, T. and R. W. Witty. 1978. Safe Programming. *BIT (Tidscrift Nordisk for Informations behandling)* 18:1-8.

ARTEWG. November 5, 1986. *Catalogue of Ada Runtime Implementation Dependencies*, draft version. Association for Computing Machinery, Special Interest Group for Ada, Ada Run-Time Environments Working Group.

Bardin, Thompson. Jan-Feb 1988. Composable Ada Software Components and the Re-Export Paradigm. *ACM Ada Letters*. VIII(1):58-79.

Bardin, Thompson. March-April 1988. Using the Re-Export Paradigm to Build Composable Ada Software Components. *ACM Ada Letters*. VIII(2):39-54.

Barnes, J. G. P. 1989. *Programming in Ada*. third edition. Reading, MA.: Addison-Wesley.

Booch, G. 1987. *Software Components with Ada - Structures, Tools and Subsystems*. Menlo Park, CA.: The Benjamin/Cummings Publishing Company, Inc.

Booch, G. 1987. *Software Engineering with Ada*. second edition. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc.

Brooks, F. B. 1975. *The Mythical Man-Month*. Essays on Software Engineering. Reading, MA:Addison-Wesley.

Charrette, R. N. 1986. *Software Engineering Environments Concepts and Technology*. Intertext Publications Inc. New York: McGraw-Hill Inc.

Cohen, N. H. 1986. *Ada as a Second Language*. New York: McGraw-Hill Inc.

Conti, R. A. March 1987. Critical Run-Time Design Tradeoffs in an Ada Implementation. *Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium*. pp. 486-495.

Cristian, F. March 1984. Correct and Robust Programs. *IEEE Transactions on Software Engineering*. SE-10(2):163-174.

Department of Defense, Ada Joint Program Office. 1984. *Rationale for the Design of the Ada Programming Language*.

Department of Defense, Ada Joint Program Office. January 1983. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A.

Foreman, J. and J. Goodenough. May 1987. *Ada Adoption Handbook: A Program Manager's Guide*. Version 1.0, CMU/SEI-87-TR-9 ESD-TR-87-110. Software Engineering Institute.

Gary, B. and D. Pokrass. 1985, *Understanding Ada A Software Engineering Approach*. John Wiley & Sons.

Goodenough, J. B. March 1986. A Sample of Ada Programmer Errors. *Unpublished draft resident in the Ada Repository under file name* PD2:<ADA.EDUCATION>PROGERRS.DOC.2.

Herr, C. S. August 1987. Compiler Validation and Reusable Software. St. Louis: a Report from the CAMP Project, McDonnell Douglas Astronautics Company.

International Workshop on Real-Time Ada Issues. 1987. *ACM Ada Letters*. VII(6). Mortonhampstead, Devon, U.K.

International Workshop on Real-Time Ada Issues II. 1988. *ACM Ada Letters*. VIII(6). Mortonhampstead, Devon, U.K.

Kernighan, B. and P. J. Plauger, 1978. The Elements of Programming Style. New York: McGraw-Hill, Inc.

Matthews, E. R. September, October 1987. Observations on the Portability of Ada I/O. *ACM Ada Letters*. VII(5):100-103.

MacLaren, L. November 1980. Evolving Toward Ada in Real Time Systems. *ACM Sigplan Notices*. 15(11):146-155.

Melliar-Smith, P. M. and B. Randell. March 1987. Software Reliability: The Role of Programmed Exception Handling. *ACM Sigplan Notices*. 12(3):95-100 .

Mowday, B. L. and E. Normand. November 1986. *Ada Progr[illegible]ming Standards*. General Dynamics Data Systems Division Departmental Instruction 414.717.

NASA. May 1987. *Ada Style Guide*. Version 1.1, SEL-87-002. Goddard Space Flight Center: Greenbelt, MD 20771.

Nissen, J. C. D., P. Wallis, B. A., Wichmann, et al. 1982. Ada-Europe Guidelines for the Portability of Ada Programs. *ACM Ada Letters*. I(3):44-61.

Nissen, J. and P. Wallis. 1984. *Portability and Style in Ada*. Cambridge University Press.

Pappas, F. March 1985. *Ada Portability Guidelines*. DTIC/NTIS #AD-A160 390.

Pyle, I. C. 1985. *The Ada Programming Language*. second edition. UK:Prentice-Hall International.

Rosen, J. P. November, December 1987. In Defense of the 'Use' Clause. *ACM Ada Letters*. VII(7):77-81.

Ross, D. March-April 1989. The Form of a Passive Iterator. *ACM Ada Letters*. IX(2):102-105.

Rymer, J. and T. McKeever. September 1986. *The FSD Ada Style Guide*. IBM Federal Systems Division Ada Coordinating Group.

Schneiderman, B. 1986. Empirical Studies of Programmers: The Territory, Paths and Destinations. *Empirical Studies of Programmers*. ed. E. Soloway and S. Iyengar. pp. 1-12. Norwood, NJ: Ablex Publishing Corp.

SofTech Inc. December 1985. *ISEC Reusability Guidelines*. Report 3285-4-247/2. also US Army Information Systems Engineering Command. Waltham MA.

Soloway, E., J. Pinto, S. Fertig, S. Letovsky, R. Lampert, D. Littman, K. Ewing. December 1986. Studying Software Documentation From A Cognitive Perspective: A Status Report. *Proceedings of the Eleventh Annual Software Engineering Workshop*. Report SEL-86-006, Software Engineering Laboratory. Greenbelt, Maryland:NASA Goddard Space Flight Center.

Stark M. and E. Seidewitz. March 1987. Towards A General Object-Oriented Ada Lifecycle. In *Proceedings of the Joint Ada Conference*, Fifth National Conference on Ada Technology and Washington Ada Symposium. 213–222.

St.Dennis, R. May 1986. *A Guidebook for Writing Reusable Source Code in Ada* –Version 1.1. Report CSC–86–3:8213. Golden Valley, Minnesota: Honeywell Corporate Systems Development Division.

United Technologies. February 9, 1987. *CENC Programmer's Guide*. Appendix A Ada Programming Standards.

VanNeste, K.F. January/February 1986. Ada Coding Standards and Conventions. *ACM Ada Letters*. VI(1):41–48.

Volz, R. A., Mudge, Naylor and Mayer. May 1985. Some Problems in Distributing Real-time Ada Programs Across Machines. *Ada in Use, Proceedings of the Ada International Conference*. pp. 14–16. Paris.

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

How did you learn of the ADA QUALITY AND STYLE GUIDE?

- ☐ Received info from company distribution point
- ☐ Heard a Consortium Technical Presentation
- ☐ Read about it in the Consortium *Quarterly*
- ☐ Saw Information on a Bulletin Board
- ☐ Read article in a journal
- ☐ Heard about it at a (non-Consortium) conference
- ☐ A fellow engineer told me
- ☐ Other _____

What kind of project will the ADA QUALITY AND STYLE GUIDE be used on? (check all that apply)

EVALUATION	IR&D	PROPOSAL	CONTRACT	INTERNAL STANDARDS
☐	☐	☐	☐	☐

How soon will you use the ADA QUALITY AND STYLE GUIDE?

IMMEDIATELY	within 3 months	within 6 months	when our proposal wins	waiting on the RFP
☐	☐	☐	☐	☐

What kind of problem will the ADA QUALITY AND STYLE GUIDE be used on? Describe

_____

_____

Estimated size of the system in lines of code

less than 10K	10K to 100K	100K to 500K	500K to 1M	Above 1M
☐	☐	☐	☐	☐

Estimated number of users

15 to 20	20 to 50	50 to 100	100 to 250	more than 250
☐	☐	☐	☐	☐

ADA QUALITY AND STYLE GUIDE REGISTRATION FORM

(Please fill out to receive updates and other information)

NAME _____ PHONE _____

P.O. BOX _____ MAIL STOP _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

COMPANY _____

ORGANIZATION _____

version 01.00.00

PLEASE COMPLETE THIS REGISTRATION FORM.

FOLD, STAPLE, AND MAIL TO RECEIVE UPDATES TO THE ADA QUALITY AND STYLE GUIDE

We are working continually to improve the ADA QUALITY AND STYLE GUIDE. Please help by completing the questions (on the reverse side) to tell us how you plan to use it.

Thank you,

THE ADA METHODS TEAM

FOLD HERE

FOLD HERE